**Міністерство освіти і науки України**
**Київський національний університет імені Тараса Шевченка**

**Факультет інформаційних технологій**
**Кафедра кібербезпеки та захисту інформації**

**ДОПУСТИТИ ДО ЗАХИСТУ:**
В.о. завідувача кафедри
кібербезпеки та захисту інформації
________ Іван ПАРХОМЕНКО
«____» червня 2023 р.

# ПОЯСНЮВАЛЬНА ЗАПИСКА

## кваліфікаційної роботи

**галузь знань** 12 Інформаційні технології
(шифр і назва галузі знань)

**спеціальність** 125 Кібербезпека
(код і назва спеціальності)

**освітній ступень** бакалавр

**освітня програма** Кібербезпека
(назва освітньо-професійної програми)

**на тему:** Оцінка методів захисту програмних комплексів інформаційно-комунікаційної системи

**Виконавець:** студент IV курсу, групи КБ-41

_______________ **Вадим МОГИЛЕВИЧ**
(підпис) (ім’я, прізвище)

| | **Ім’я ПРІЗВИЩЕ** | **Підпис** |
|---|---|---|
| **Керівник** | Сергій ДАКОВ | |

| | | |
|---|---|---|
| **Нормоконтроль** | Інна МИХАЛЬЧУК | |

**Київ 2023**

**Міністерство освіти і науки України**
**Київський національний університет імені Тараса Шевченка**

**Факультет інформаційних технологій**
**Кафедра кібербезпеки та захисту інформації**

**ЗАТВЕРДЖЕНО:**
В.о. завідувача кафедри кібербезпеки
та захисту інформації
________ Сергій ТОЛЮПА
«24» жовтня 2022 р.

# ЗАВДАННЯ
## на виконання кваліфікаційної роботи

**спеціальності** 125 Кібербезпека
(код і назва спеціальності)
**освітньої програми** Кібербезпека
(назва освітньо-професійної програми)

**Студенту** **КБ-41** **Могилевичу Вадиму Дмитровичу**
(група) (прізвище ім'я по батькові)

**Тема кваліфікаційної роботи** Оцінка методів захисту програмних комплексів інформаційно-комунікаційної системи

**1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ**

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №3 від 20.10.2022 р.

**2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ**

Архітектура і функціонування інформаційно-комунікаційної системи, методи захисту програмних комплексів.

**3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ**

Необхідно ознайомитися з функціонуванням програмних комплексів ІКС, проаналізувати типи атак на операційні системи та методи захисту.

**4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ**

**Практична цінність** Розроблена програмна реалізація методів захисту програмних комплексів ІКС для подальшого дослідження та аналізу.

**5. ДАТА ВИДАЧІ ЗАВДАННЯ**

Дата видачі завдання: 24 жовтня 2022 року

| | | |
|---|---|---|
| Завдання видав | ______________ (підпис) | Сергій ДАКОВ (ім'я, прізвище) |
| Завдання прийняв до виконання | ______________ (підпис) | Вадим МОГИЛЕВИЧ (ім'я, прізвище) |

**КАЛЕНДАРНИЙ ПЛАН**

| № п/п | Найменування етапів робіт | Строки виконання робіт (початок-кінець) | Відмітка про виконання |
|---|---|---|---|
| 1 | Уточнення постановки задачі | 24.10.2022 – 22.01.2023 | *виконано* |
| 2 | Аналіз літератури | 23.01.2023 – 11.02.2023 | *виконано* |
| 3 | Обґрунтування вибору рішення | 12.02.2023 – 15.02.2023 | *виконано* |
| 4 | Дослідження програмних комплексів та загроз інформаційно-комунікаційній системі | 16.02.2023 – 04.03.2023 | *виконано* |
| 5 | Дослідження особливостей захисту ІКС на рівні операційної системи | 05.03.2023 – 21.03.2023 | *виконано* |
| 6 | Аналіз методів захисту, що використовуються для захисту від атак переповнення буфера | 22.03.2023 – 08.04.2023 | *виконано* |
| 7 | Спрощена програмна реалізація методів захисту програмних комплексів ІКС | 09.04.2023 – 10.05.2023 | *виконано* |
| 8 | Оформлення пояснювальної записки | 11.05.2023 – 27.05.2023 | *виконано* |
| 9 | Підготовка до захисту кваліфікаційної роботи | 28.05.2023 – 12.06.2023 | *виконано* |

| | | |
|---|---|---|
| Завдання видав | ______________ (підпис) | Сергій ДАКОВ (ім'я, прізвище) |
| Завдання прийняв до виконання | ______________ (підпис) | Вадим МОГИЛЕВИЧ (ім'я, прізвище) |

Термін подання кваліфікаційної роботи до ЕК 12 червня 2023 року

# РЕФЕРАТ

Кваліфікаційна робота складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел, додатків, має 66 сторінок основного тексту. Список використаних джерел містить 49 найменування і займає 7 сторінок.

**Методи дослідження** кваліфікаційної роботи:

аналіз літератури;

аналіз документів;

порівняння;

**Об'єктом дослідження** є процеси функціонування програмних комплексів інформаційно - комунікаційної системи.

**Предметом дослідження** в даній роботі є методи захисту програмних комплексів інформаційно - комунікаційної системи.

**Метою роботи** є оцінка методів захисту програмних комплексів інформаційно-комунікаційної системи на основі аналізу сучасних атак та методів захисту операційної системи.

Для досягнення поставленої мети необхідно вирішити наступні взаємопов’язані завдання:

1. Аналіз програмних комплексів та загроз інформаційно-комунікаційній системі;
2. Аналіз особливостей захисту ІКС на рівні ОС;
3. Аналіз методу рандомізації розміщення адресного простору;
4. Аналіз методу цілісності потоку керування;
5. Порівняння між методом цілісності потоку керування та методом рандомізації розміщення адресного простору для захисту від атак переповнення буфера.

**Ключові слова:** захист інформаційно-комунікаційної системи, програмний комплекс, атаки на операційні системи, метод *ASLR*, метод *CFI*.

## ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

| | | |
|---|---|---|
| ASLR | – | Address space layout randomization |
| CFI | – | Control flow integrity |
| CPU | – | Central processing unit |
| DEP | – | Data execution prevention |
| MMU | – | Memory management unit |
| PIE | – | Position independent executable |
| ROP | – | Return oriented programming |
| SMAP | – | Supervisor mode access prevention |
| SMEP | – | Supervisor mode execution prevention |
| АК | – | Апаратний комплекс |
| АПК | – | Апаратно-програмний комплекс |
| ІКС | – | Інформаційно-комунікаційна система |
| МРРТС | – | Метод рандомізації розташування таблиць сторінок |
| ОС | – | Операційна система |
| ПЗ | – | Програмне забезпечення |
| ПК | – | Програмний комплекс |
| ЕОМ | – | Електронна обчислювальна машина |
| ЕКО | – | Електронне комунікаційне обладнання |

# ЗМІСТ

ВСТУП.......... 8
РОЗДІЛ 1 АНАЛІЗ ПРОГРАМНИХ КОПЛЕКСІВ ТА ЗАГРОЗ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНІЙ СИСТЕМІ.......... 10
1.1 Аналіз інформаційно-комунікаційної системи.......... 10
1.2 Класифікація програмних засобів та загроз інформаційній безпеці інформаційно-комунікаційної системи.......... 14
1.2.1 Класифікація програмних засобів.......... 14
1.2.2 Класифікація загроз інформаційній безпеці в інформаційно-комунікаційній системі.......... 19
1.3 Постановка завдання.......... 21
Висновок до першого розділу.......... 22
РОЗДІЛ 2 ДОСЛІДЖЕННЯ ОСОБЛИВОСТЕЙ ЗАХИСТУ ІКС НА РІВНІ ОПЕРАЦІЙНОЇ СИСТЕМИ.......... 24
2.1 Способи захисту ОС та вразливості, що пов'язані з пошкодженням пам'яті.. 24
2.1.1 Розмежування рівнів привілеїв.......... 24
2.1.2 Традиційні моделі загроз.......... 27
2.2 Атаки з пошкодженням пам'яті.......... 29
2.2.1 Причини пошкодження пам'яті.......... 29
2.2.2 Експлойт пошкодження пам'яті.......... 30
2.3 Сучасні атаки та захист.......... 32
2.3.1 Захист від повторного використання коду.......... 32
2.3.2 Атаки лише на дані ядра ОС.......... 37
2.3.3 Способи захисту від атаки на дані ядра ОС.......... 38
Висновок до другого розділу.......... 41
РОЗДІЛ 3 ОЦІНКА МЕТОДІВ ЗАХИСТУ ПРОГРАМНИХ КОМПЛЕКСІВ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНОЇ СИСТЕМИ.......... 42
3.1 Метод рандомізації розміщення адресного простору (ASLR).......... 42

3.2 Метод цілісності потоку керування (CFI) ........ 46
3.2.2 Варіації методів CFI, для запобігання перенаправлення потоку керування.. 49
3.3 Порівняння рівня безпеки між методами цілісності потоку керування і рандомізацією адресного простору для захисту від атак переповнення буфера в операційній системі Windows 10 ........ 56
Висновок до третього розділу ........ 57
ВИСНОВКИ ........ 59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ........ 60
ДОДАТОК А. Список опублікованих праць за темою кваліфікаційної роботи........ 67
ДОДАТОК Б. Лістинг програми ........ 69

# ВСТУП

**Актуальність** даної роботи визначається тим, що на теперішній час інформаційно-комунікаційні системи не тільки є основою системи управління органів державної влади України, а також широко використовуються в різних галузях виробництва. Досвід бойових дій з російською федерацією також показав, що ІКС в першу чергу зазнають різного типу атак.

Аналіз сучасного стану існуючих ІКС показав, що вони не повною мірою відповідають вимогам інформаційної безпеки. В опублікованих науково-технічних роботах по дослідженню ІКС відсутній єдиний підхід і теоретичні результати, які дозволили б комплексно враховувати основні фактори, які впливають на інформаційну безпеку. Одним з таких важливих факторів є програмні комплекси (сукупності програмних засобів) в складі ІКС, які мають обмежену інформаційну безпеку та здійснюють суттєвий вплив на ефективність функціонування ІКС, причому цей вплив в перспективі буде зростати. Все це ускладнює прийняття ефективних науково-обґрунтованих рішень при виконанні комплексу задач, направлених на забезпечення необхідного рівня інформаційної безпеки ІКС та підвищення їх можливостей.

Тому **метою роботи** є оцінка методів захисту програмних комплексів інформаційно - комунікаційної системи на основі аналізу сучасних атак та методів захисту операційної системи.

Для досягнення поставленої мети необхідно вирішити наступні взаємопов’язані *завдання*:

1. Аналіз програмних комплексів та загроз інформаційно-комунікаційній системі;
2. Аналіз особливостей захисту ІКС на рівні ОС;
3. Аналіз методу рандомізації розміщення адресного простору;
4. Аналіз методу цілісності потоку керування;

5. Порівняння між методом цілісності потоку керування та методом рандомізації розміщення адресного простору для захисту від атак переповнення буфера.

**Об'єктом дослідження** є процеси функціонування програмних комплексів інформаційно - комунікаційної системи.

**Предметом дослідження** в даній роботі є методи захисту програмних комплексів інформаційно - комунікаційної системи.

**Методи дослідження** кваліфікаційної роботи:

аналіз літератури;

аналіз документів;

порівняння.

# РОЗДІЛ 1
# АНАЛІЗ ПРОГРАМНИХ КОПЛЕКСІВ ТА ЗАГРОЗ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНІЙ СИСТЕМІ

Метою даного розділу є аналіз інформаційно-комунікаційної системи, особливості її функціонування, а також класифікація програмних засобів та визначення загроз ІКС. Результати цього аналізу дають можливість обґрунтувати актуальність тематики даної роботи, визначити основні поняття об’єкта та предмета дослідження, а також сформулювати мету і наукову задачу даного дослідження.

## 1.1 Аналіз інформаційно-комунікаційної системи

Згідно з [1] інформаційно-комунікаційна система - сукупність інформаційних та електронних комунікаційних систем, які у процесі обробки інформації діють як єдине ціле.

Інформаційно-комунікаційна система (ІКС) є комплексом технічних засобів, програмного забезпечення та організаційних процесів, призначених для обміну інформацією між користувачами за допомогою електронних комунікаційних мереж. Ця система дозволяє збирати, зберігати, обробляти, передавати і отримувати інформацію в електронному вигляді (рис. 1.1).

В основі побудови сучасних електронних комунікаційних мереж лежить концепція мультисервісних мереж. Мультисервісна мережа - єдина інформаційно-комунікаційна структура, що підтримує всі види трафіку (дані, голос, відео) і надає всі види послуг (традиційні та нові, базові та додаткові) у будь-якій точці, у будь-який час, у будь-якому наборі та обсязі, з диференційованою гарантованою якістю обслуговування, що задовольняє різні категорії користувачів [2].

Розглянемо відмінні риси концепції мультисервісності, які стосуються різних сторін побудови мережі:

1. Конвергенція завантаження мережі, що визначає передачу різних типів

трафіку в рамках єдиного формату представлення даних. Конвергенція завантаження мережі визначає тенденцію використання мереж з комутацією пакетів для передавання і аудіо-, і відеопотоків, і власне даних мереж. Однак це не заперечує вимоги диференціювання трафіку відповідно до наданої якості послуг.

Рисунок 1.1 – Варіант побудови ІКС

2. Конвергенція протоколів, що визначає перехід від безлічі існуючих мережевих протоколів до загального, як правило, *IP*-протоколу. Мультисервісні мережі орієнтуються на єдиний протокол і різні сервіси, необхідні для підтримки різних типів трафіку.

3. Фізична конвергенція, що визначає передачу різних типів трафіку в рамках єдиної мережевої інфраструктури.

4. Конвергенція пристроїв, що визначає тенденцію побудови архітектури мережевих пристроїв, здатної в рамках єдиної системи підтримувати різнотипний трафік.

5. Конвергенція додатків, що визначає інтеграцію різних функцій у рамках єдиного програмного засобу. Наприклад, *Web*-браузер дає змогу об'єднати в рамках однієї сторінки мультимедіа-дані типу звукового, відеосигналу, графіки високої роздільної здатності тощо.

6. Організаційна конвергенція, що передбачає централізацію служб (мережевих, комунікаційних, інформаційних).

Усі перелічені вище особливості визначають різні сторони побудови мультисервісної мережі, що здатна передавати трафік різного типу як у периферійній частині мережі, так і в її ядрі (рис. 1.2). В основі архітектури мультисервісної мережі лежить принцип відокремлення одна від одної функцій перенесення і комутації, функцій управління викликом і функцій управління послугами, які реалізуються зазначеними вище функціональними рівнями.

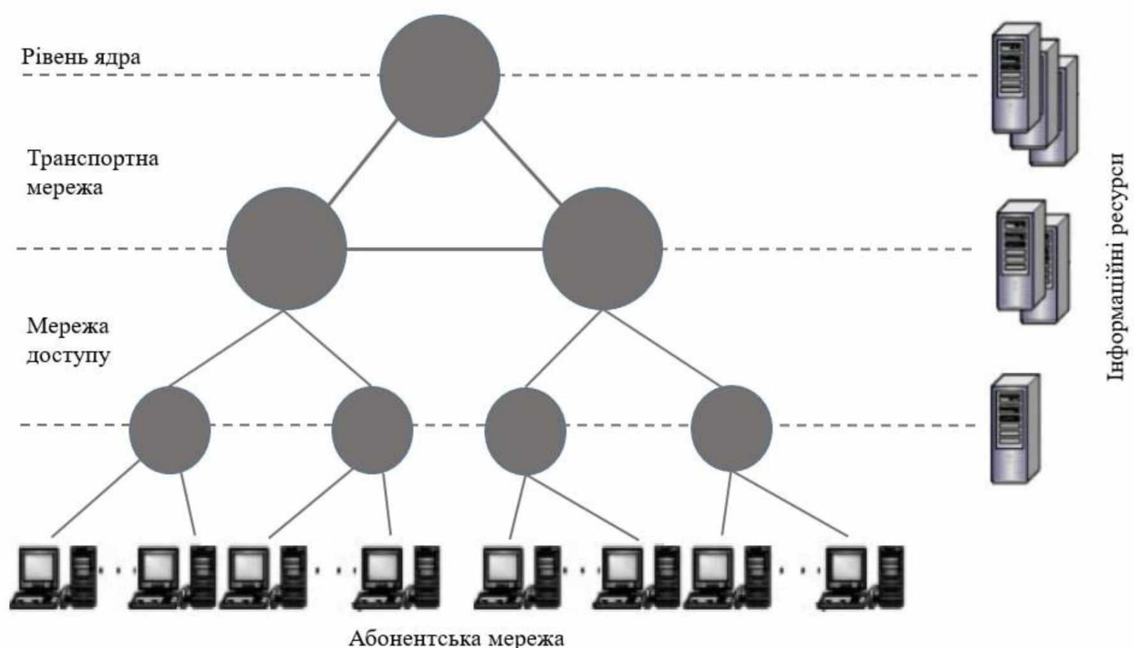



Рисунок 1.2 – Варіант структури мультисервісної мережі

Отже, мультисервісну мережу можна представити, як сукупність транспортної мережі, мереж доступу та абонентських мереж. На кожному з цих рівнів використовується комунікаційне обладнання в комплексі зі своїм програмним забезпеченням, що задовольняє вимогам кожного з цих рівнів мультисервісної мережі зв’язку.

Інформаційно-комунікаційна система включає в себе різноманітні елементи, такі як комп'ютери, мережеве обладнання (маршрутизатори, комутатори, каналоутворююче обладнання, тощо), програмне забезпечення для обробки інформації, бази даних, електронні пристрої зв'язку (телефони, смартфони, планшети, ЕОМ) та інші пристрої. Вони утворюють інфраструктуру для обміну даними та комунікації між користувачами.

Основні завдання ІКС включають:

- забезпечення обміну інформацією: ІКС дозволяє передавати дані між користувачами з різних місць і у різний час. Це може бути передача текстової, голосової, відео-або графічної інформації;
- забезпечення доступу до інформації: ІКС дає можливість отримувати інформацію з різних джерел, таких як інтернет, бази даних, електронні документи, тощо;
- забезпечення обробки інформації: ІКС надає інструменти для обробки інформації, такі як програми для аналізу даних, створення звітів, редагування документів, тощо;
- забезпечення комунікації: ІКС дозволяє спілкуватися і взаємодіяти між користувачами за допомогою різних засобів комунікації.

Результати проведеного аналізу процесів функціонування існуючих і перспективних ІКС дозволяє представити обладнання, що використовується, у вигляді складного апаратно-програмного комплексу (АПК), що складається з двох взаємопов’язаних підсистем:

1) апаратного комплексу (АК) – сукупності технічних засобів, що включають в свій склад апаратуру каналоутворення, пристрої комутації і маршрутизації, сервери, апаратуру *IP*-шифрування, кінцеве та інше обладнання;

2) програмного комплексу (ПК) – сукупності програмних засобів («м’якого» обладнання АПК), в складі якого зазвичай виділяють три частини: математичне забезпечення (сукупність математичних моделей, методів і алгоритмів), інформаційне забезпечення (бази даних з системами управління, файлові структури з каталогами, константи та ін. елементи) та програмне забезпечення (системне та функціональне) [3].

Під програмним засобом будемо розуміти об’єкт, який складається з програм, процедур, правил, а також, якщо передбачено, супутніх їм документації і даних, що відносяться до функціонування обладнання.

Аналіз показав, що ПЗ є найбільш розвинутою за структурою та функціональним зв’язкам складовою частиною апаратно-програмних комплексів обладнання. Більш того, при складній взаємодії технічних і програмних засобів часто важко ідентифікувати першоджерело порушення нормального функціонування обладнання. Тому важливо не тільки забезпечити високу безпеку ПЗ, але й врахувати її при оцінці безпеки обладнання ІКС загалом.

Таким чином, програмний комплекс, є важливою складовою частиною обладнання, який має обмежену безпеку та суттєво впливає на безпеку ІКС. Цей вплив має тенденцію зростати в перспективі у міру ускладнення задач, що вирішуються ІКС.

## 1.2 Класифікація програмних засобів та загроз інформаційній безпеці інформаційно-комунікаційної системи

Проведемо класифікацію програмних засобів та визначимо загроз інформаційній безпеці інформаційно-комунікаційної системи.

### 1.2.1 Класифікація програмних засобів

Важливою складовою інформаційно-комунікаційної системи є програмні засоби (забезпечення), яке можна розподілити на системне, прикладне та інструментальне.

Системне програмне забезпечення – сукупність програм, які забезпечують роботу, керування та контролюють доступ к апаратної частині електронних обчислювальних машин (серверне обладнання, персональні обчислювальні машини), а також електронного комунікаційного обладнання (маршрутизатори, комутатори, каналоутворююче обладнання, обладнання *IP* шифрування та ін.)

Системне ПЗ забезпечує загальну організацію процесу обробки інформації, керує ресурсами ЕОМ та ЕКО, а також забезпечує взаємодію та інтерфейси між ЕКО (ЕОМ) користувачем, апаратною та програмною частинами.

До системного програмного забезпечення можна віднести: операційні системи, оболонки операційних систем, драйвери, архіватори, антивірусні програми, програми обслуговування дисків, програми обслуговування комп'ютерних мереж тощо (рис. 1.3).

Системне ПЗ є невід'ємною частиною ЕОМ та ЕКО незалежно від специфіки предметної області та розв'язуваних ними завдань.

Окремо необхідно виділити системне програмне забезпечення ЕКО (наприклад, ПЗ фірми *Cisco* - *Cisco IOS*, *Cisco IOS XE*, *Cisco IOS XR* и *Cisco NX-OS*, ПЗ фірми *Mikrotik* - *RouterOS* та ін.).

Рисунок 1.3 – Види системного програмного забезпечення

В подальшому дослідженні будемо розглядати різні типи атак та методи захисту операційних систем. Це пов'язано з тим, що операційні системи є важливою частиною ЕОМ та ЕКО. В табл. 1.1 представлено операційні системи ЕКО провідних виробників, що використовується в ІКС.

*Таблиця 1.1*

Операційні системи ЕКО

| Виробник | Назва ЕКО | ОС |
|---|---|---|
| *Cisco* | *Catalyst* 6500; *C*9500-48*Y*4*C*-*E*; *C*9300-24*S*-*E*; *SF*350-48-*K*9-*EU*. | *Cisco IOS* |
| *Huawei* | *S*1720-52*GWR*-4*P*; *S*5720*S*-28*P*-*SI*-*AC*; *S*5700*S*-52*P*-*LI*-*AC*; *S*1730S-*H*24*T*4*S*-*A*. | *Huawei VRP* |
| *Ubiquiti* | *EdgeSwitch ES*-48; *UniFi Switch US*-48-500*W*. | *EdgeOS* |
| *Mikrotik* | *RB*1100*AH*x4 *Dude Edition*; *CRS*328-24*P*-4*S*. | *RouterOS* |
| *H3C* | *S*12500*X*-*AF*; *S*9850; *S*9820; *S*6890; *S*10500*X*; *S*6520*X*-*HI*; *S*5560*S*-*EI*; *S*5130*S*-*HI*; *S*5000*V*3-*EI*. | *Comware V7* |
| *Siemens* | *RUGGEDCOM RST*2228*P* | *MindSphere* |
| *Alcatel* | *OmniSwitch* 6350-24; *OS*6850*EP*48*X*; *OS*6465-*P*6. | *AOS* |

Прикладне програмне забезпечення - це комплекс взаємопов'язаних програм, що призначені для вирішення завдань певного класу конкретної предметної області. У зв'язку з величезною різноманітністю прикладного програмного забезпечення існує безліч варіантів його класифікації. Аналіз науково-технічної літератури показав доцільність представити прикладне програмне забезпечення наступним чином:

- прикладними програми загального призначення;
- прикладними програми спеціального (професійного) призначення.

Прикладні програми загального призначення це програмними засобами для вирішення функціональних завдань і є найчисленнішим класом програмних продуктів. До прикладних програм загального призначення відносять програми, які можуть застосовуватися в різних галузях людської діяльності для опрацювання текстів, малюнків, баз даних, електронних таблиць, створення презентацій тощо. Класифікація прикладних програм загального призначення наведено на рис. 1.4.

Прикладні програми спеціального призначення використовуються для реалізації завдань опрацювання даних у певній галузі діяльності, на конкретному підприємстві, в організації, фірмі або їх підрозділі. До такого типу програм відносять програми для створення відеоефектів при виробництві кінофільмів, креслень машин

і механізмів у конструкторських і проектних бюро, діагностування захворювань у медичних закладах, створення шкільного розкладу уроків тощо.

Рисунок 1.4 – Класифікація прикладних програм загального призначення

Класифікація прикладних програм загального призначення наведено на рис. 1.5.

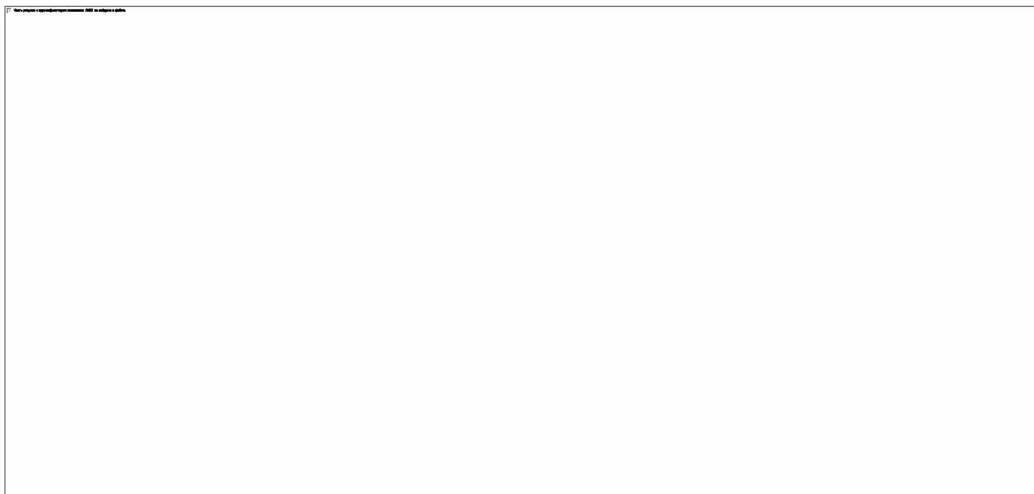


Рисунок 1.5 – Класифікація прикладних програм спеціального призначення

Інструментальне програмне забезпечення відноситься до програмних пакетів, призначених для створення інші програми. Варіант класифікації інструментального програмного забезпечення наведено в табл. 1.2.

Інтегровані середовища програмування – система розробки програмного забезпечення, включає основні види інструментального ПЗ: спеціалізований текстовий редактор, транслятор, компоновщик, відладчик і бібліотеки. Прикладами

інтегрованих програмних середовищ є *Turbo Paskal*, *Quick Basic*, *Borland C++* (для розробки консольних програм), *Microsoft Visual C++*, *Microsoft Visual Basic*, *Embarcadero Delphi*, *Embarcadero JBuilder* (для розробки віконних додатків *Windows*).

*Таблиця 1.2*

Класифікації інструментального програмного забезпечення

| | Вид інструментального ПЗ | Призначення | Приклади |
|---|---|---|---|
| 1. | Спеціалізовані текстові редактори | для створення та редагування коду програми | *Notepad++* |
| 2. | Транслятори: | для перетворення програми в машинний код | |
| | асемблери | для перетворення програми мовою Асемблер | *Macro* Assemblei(*MASM*), *Turbo Assembler*(*TASNf*) - для процесорів x86 |
| | компілятори | для перетворення програм написаних мовами високого рівня (Паскаль. Делфі. Сі. Бейсік). Перетворення відбувається повністю, одноразово, зі створенням файлу | *ONU Compiler Collection* (*GCC*)- для Сі. *C++*. *Java*. *Fortran* та ін,; *Free Pascal Compiler* (*FPS*)- для Паскаля: *Intel C++ compiler* (для *C*, *C++*. *Fortran*) |
| | інтерпретатори | для покомандного перетворення та виконання програми мовою | |
| 3. | Компоновщики (лінкерн. редактори зв'язку) | для збирання виконуваного файлу з об'єктних файлів (двійкові коди окремих файлів програми) | |
| 4. | Налагоджувані (дебагери) | дозволяють покроково виконувати програму, передивлятися та змінювані значення змінних у ході виконання програми та ін. | |
| 5. | Бібліотеки | містять процедури та функції, які можуть використовувати програмісти у своїх програмах | |

### 1.2.2 Класифікація загроз інформаційній безпеці в інформаційно-комунікаційній системі

Класифікація загроз інформаційній безпеці може бути здійснена за багатьма ознаками.

Під загрозою інформаційній безпеці системи розуміємо можливість реалізації впливу на інформацію, що призводить до порушення конфіденційності, цілісності або доступності даних, а також можливість впливів на компоненти системи, які можуть призводити до втрати або знищення інформації чи збою функціонування ІКС.

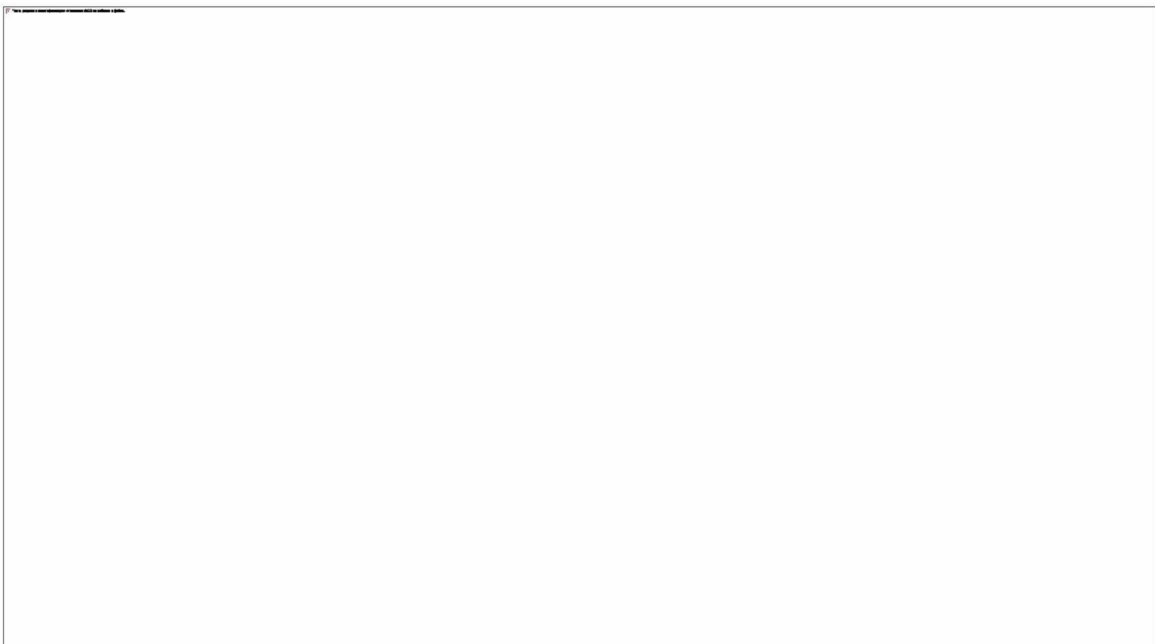


Рисунок 1.6 – Варіант класифікації загроз інформаційній безпеці ІКС

На рис. 1.6 наведено варіант класифікації загроз інформаційній безпеці.

1. За природою виникнення:

- природні – це небезпеки, які виникають в результаті природних явищ і процесів, таких як землетруси, повені, тайфуни, цунамі, пожежі тощо. Ці явища можуть виникати безпосередньо через природні процеси або бути наслідком глобальних змін клімату. Особливість – складність їх прогнозування;
- штучні загрози – виникають в результаті дії людського фактора.

2. За ступенем навмисності загрози:

- випадкові – можуть виникати через ненавмисні помилки або халатність персоналу (введення помилкових даних або ненавмисне псування устаткування);

- навмисні – виникають, коли зловмисник цілеспрямовано виконує дії з метою нанесення шкоди системі, мережі або її користувачам. Наприклад, SQL-ін'єкція є однією з найбільш поширених навмисних загроз, яка використовується для зміни або вилучення важливих даних з баз даних за допомогою вставки шкідливого коду SQL в веб-форми або параметри запитів до баз даних.

3. Залежно від джерела загрози:

- природне середовище. Наприклад, блискавки, повені, різке підвищення (пониження) температури атмосфери;
- людина, як джерело загроз. Наприклад, влаштування довірених осіб недержавною організацією на посади, які займаються обслуговуванням державних інформаційних систем, може створити загрозу для безпеки цих систем;
- санкціоновані програмно-апаратні засоби. Наприклад, некомпетентне використання системних утиліт;
- несанкціоновані програмно-апаратні засоби. Наприклад, установка в систему кейлогерів, а також використання особистих флешок, мобільних телефонів та ін., які можуть містити шкідливе програмне забезпечення.

4. За положенням джерела загрози:

- джерело розташоване зовні контрольованої зони. Наприклад, перехоплення даних, що передаються каналами зв’язку, акустичної інформації, побічних електромагнітних випромінювань;
- джерело розташоване в межах контрольованої зони. Наприклад, несанкціоноване використання диктофонів або розкрадання носіїв, що містять конфіденційну інформацію.

5. За ступенем впливу на системи:

- пасивні загрози – можуть означати потенційні атаки на систему, які не завдають безпосередньої шкоди інформації або структурі системи. Зазвичай такі загрози є менш очевидними, але їх наслідки можуть бути значними, якщо не вжити заходів для їх запобігання. Наприклад, несанкціоноване копіювання файлів з даними;
- активні загрози – включають активну дію з боку зловмисників. Наприклад, моніторинг мережевого трафіку для перехоплення паролів та даних.

6. За способом доступу до ресурсів ІКС:

- стандартний доступ. Наприклад, шантаж, підкуп, фізичне насильство щодо законного власника з метою несанкціонованого отримання паролю;
- нестандартний доступ. Наприклад, використання незадекларованих можливостей засобів захисту.

На практиці використовується класифікація загроз (рис. 1.7), яка ґрунтується на порушенні цілісності інформації (будь-яке зловмисне спотворення інформації), порушення конфіденційності інформації (інформація розкривається третім особам без згоди власника інформації), порушення доступності інформації (інформацію неможливо отримати або вона недоступна в той момент, коли вона потрібна).

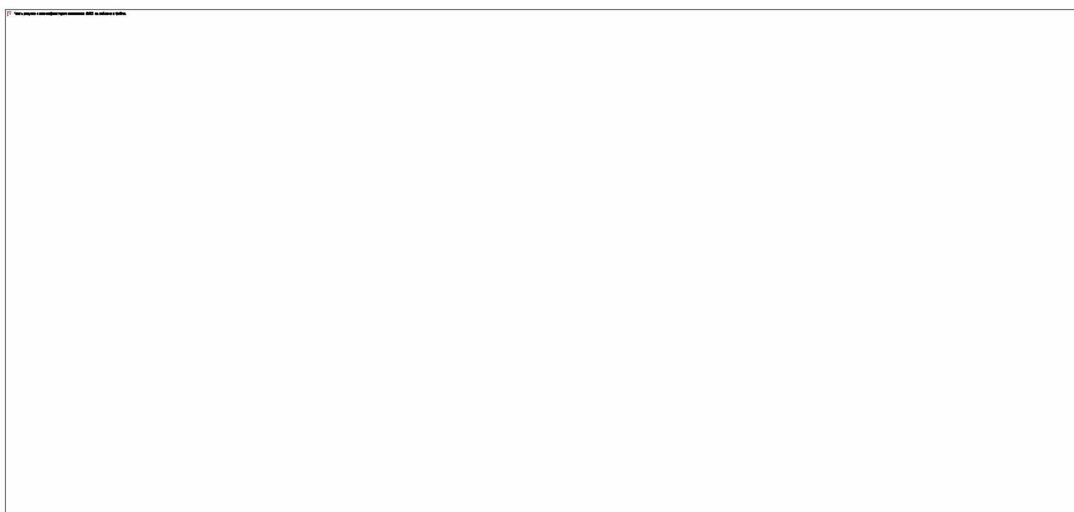


Рисунок 1.7 – Класифікація загроз

## 1.3 Постановка завдання

Проведений аналіз стану існуючих ІКС показав, що вони не повною мірою відповідають вимогам інформаційної безпеки [1]. В опублікованих науково-технічних роботах по дослідженню ІКС відсутній єдиний підхід і теоретичні результати, які дозволили б комплексно враховувати основні фактори, які впливають на інформаційну безпеку. Одним з таких важливих факторів є програмні комплекси (сукупності програмних засобів) в складі ІКС, які мають обмежену інформаційну безпеку, здійснюють суттєвий вплив на ефективність функціонування ІКС, причому цей вплив в перспективі буде зростати. Все це ускладнює прийняття ефективних

науково-обґрунтованих рішень при виконанні комплексу задач, направлених на забезпечення необхідного рівня інформаційної безпеки ІКС та підвищення їх можливостей.

Таким чином, можна зробити висновок про існування на теперішній час наступних протиріч:

- на практиці – між високими вимогами до інформаційної безпеки ІКС та реальної обмеженої безпекою програмного забезпечення (програмних засобів) ІКС;
- в теорії – між обмеженими можливостями відомих методів захисту інформаційної безпеки та необхідністю комплексного захисту ІКС.

Тому актуальним є наукове завдання, яке направлене на вирішення вказаних вище протиріч і полягає в аналізі сучасних атак та методів захисту операційної системи.

Метою дослідження є підвищення безпеки програмних комплексів інформаційно - комунікаційної системи на основі аналізу сучасних атак та методів захисту операційної системи.

Об'єкт дослідження – процеси функціонування програмних комплексів інформаційно - комунікаційної системи

Предмет дослідження – методи захисту програмних комплексів інформаційно - комунікаційної системи

**Висновок до першого розділу**

1. Аналіз існуючої інформаційно-комунікаційної системи дозволив зробити висновок щодо існування на теперішній час протиріч: на практиці – між високими вимогами до інформаційної безпеки ІКС та реальної обмеженої безпекою програмного забезпечення (програмних засобів) ІКС; в теорії – між обмеженими можливостями відомих методів захисту інформаційної безпеки та необхідністю комплексного захисту ІКС.

2. Вирішення цих протиріч визначається рішенням сформульованої наукової задачі – аналіз сучасних атак та методів захисту операційної системи.

3. Для рішення сформульованої наукової задачі обґрунтована сукупність основних взаємопов'язаних напрямків дослідження:

1) Аналіз програмних комплексів та загроз інформаційно-комунікаційній системі;

2) Аналіз особливостей захисту ІКС на рівні ОС;

3) Аналіз методу рандомізації розміщення адресного простору;

4) Аналіз методу цілісності потоку керування;

5) Порівняння між методом цілісності потоку керування та методом рандомізації розміщення адресного простору для захисту від атак переповнення буфера.

Вирішення комплексу взаємопов'язаних задач дослідження, які входять у вказані вище напрямки дослідження, визначає структуру і зміст наступних розділів кваліфікаційної роботи, а сукупність отриманих при цьому результатів дає рішення сформульованої наукової задачі.

# РОЗДІЛ 2
# ДОСЛІДЖЕННЯ ОСОБЛИВОСТЕЙ ЗАХИСТУ ІКС НА РІВНІ ОПЕРАЦІЙНОЇ СИСТЕМИ

Операційні системи (ОС) підтримують велику кількість програм (додатків), що працюють одночасно або з розділеним часом на одній апаратній платформ.

Оскільки додатки можуть бути вразливими до атак або навіть у деяких випадках шкідливі, сучасні операційні системи розроблені як еталонні монітори, що відокремлюють контроль доступу та керування ресурсами від процесів користувача [4]. Реалізація цього поділу вимагає апаратної підтримки, а сучасні процесори забезпечують розмежування рівнів привілеїв та віртуальної пам'яті.

## 2.1 Способи захисту ОС та вразливості, що пов’язані з пошкодженням пам’яті

### 2.1.1 Розмежування рівнів привілеїв

Програмне забезпечення ядра ОС зазвичай працює з вищими привілеями, ніж прикладне програмне забезпечення. Багато процесорів підтримують різні рівні привілеїв та дозволяють програмному забезпеченню перемикатися між ними під час виконання, наприклад, за допомогою спеціальних інструкцій передачі керування. Як правило, існує непривілейований рівень для виконання програм і привілейований рівень, що зарезервований для ОС. Коли процесор виконує процес з найвищим рівнем привілеїв, запущене програмне забезпечення має повний доступ до будь-яких інструкцій, може мати прямий доступ до апаратного забезпечення платформи, взаємодіяти з регістрами налагодження та керування платформою, отже, має повний контроль над системою. Виконання коду на непривілейованому рівні залишає лише обмежену кількість доброякісних інструкцій, доступних для програмного забезпечення, наприклад, зазвичай можливо лише завантажувати та зберігати у

непривілейовану пам’ять або регістри, обчислювати арифметичні операції, оцінювати умови та виконувати розгалуження. Під час виконання апаратне забезпечення підтримує окремі контексти виконання для різних рівнів привілеїв, а перемикання між рівнями привілеїв прозоро викликає відповідний контекст. Наприклад, коли користувальницька програма аварійно завершує роботу через нешкідливу операцію, наприклад ділення на нуль, процесор автоматично викликає код з найвищим рівнем привілеїв (тобто ОС). Після цього ОС може обробити стан помилки, спричинений програмою користувача та завершити роботу програми.

Програмне забезпечення на рівні користувача також може цілеспрямовано викликати ОС. Наприклад, якщо програма потребує привілейованого доступу, наприклад, для відкриття файлу, що зберігається на жорсткому диску платформи, вона має надіслати запит до операційної системи на доступ до диска та отримання файлу від свого імені.

Таким чином ОС може бути посередником у будь-якому доступі і забезпечувати дотримання політик безпеки. Служби ОС стандартизовані у формі системних викликів [5] , і цей інтерфейс представляє одну з основних частин поверхні атаки ядра на користувацьке програмне забезпечення.

Розглянемо захист віртуальної пам'яті. На додаток до поділу привілеїв, ОС зазвичай також підтримує  ізоляцію пам'яті для прикладного програмного забезпечення. У багатьох центральних процесорах (*CPU*) ізоляція пам'яті забезпечується спеціальним апаратним забезпеченням (блок керування пам'яттю (*MMU*)), що є частиною процесора. Якщо *MMU* вимкнено, всі звернення до пам'яті з програмного забезпечення працюють безпосередньо з фізичною пам'яттю, що означає, що ізоляція пам'яті не забезпечується. Якщо *MMU* увімкнено операційною системою, всі звернення до пам'яті з програмного забезпечення інтерпретуються як віртуальні адреси, і кожен процес асоціюється з власним віртуальним адресним простором. Така опосередкованість змушує перевіряти всі звернення з програмного забезпечення за допомогою набору контролів перед тим, як буде дозволено будь-який фактичний доступ до фізичної пам'яті.

Ці вказівники керування доступом зберігаються в ієрархічній структурі даних,

яка називається таблицею сторінок. На рис. 2.1 процес *B*

Рисунок 2.1 – Варіант схеми захисту віртуальної пам'яті на основі сторінок

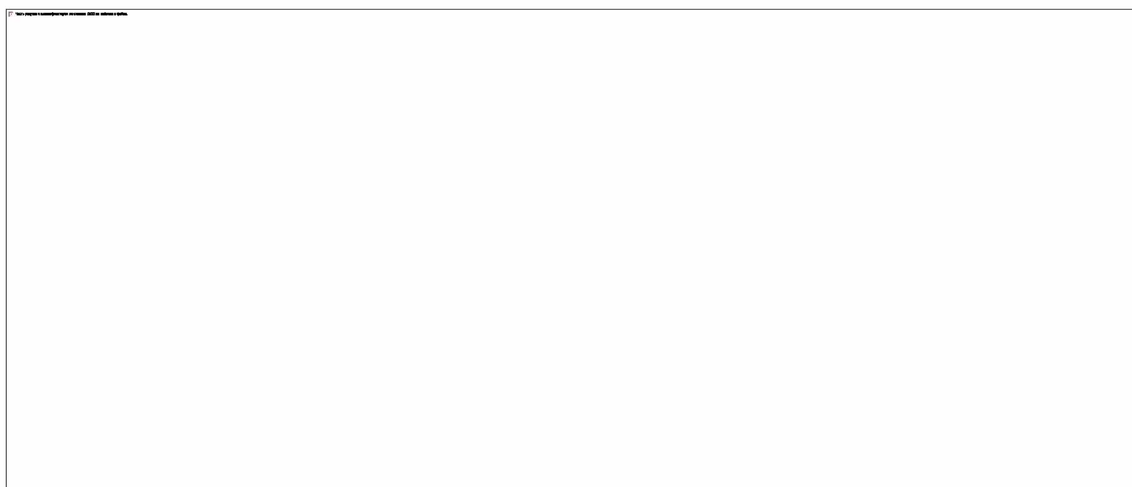
отримує доступ до віртуальної адреси на кроці 1, яка транслюється та перевіряється *MMU*. Щоб отримати елементи керування доступом для заданої віртуальної адреси, потрібно переглянути таблицю сторінок на кроці 2, що може включати кілька рівнів опосередкованості у фізичній пам’яті. Тому цей обхід сторінок вважається дорогою операцією та часто оптимізується ОС і апаратним забезпеченням різними способами. Після отримання запису таблиці сторінок для даної адреси віртуальну адресу можна перевести на кроці 3 у відповідну фізичну адресу, яка може інтерпретуватися безпосередньо шиною пам'яті платформи. Запис таблиці сторінок також зберігає прапорці керування доступом, пов’язані з цим записом, наприклад запитуваний рівень привілеїв, доступність для читання, запису чи виконання. Залежно від характеру доступу та поточного рівня привілеїв коду, який надав доступ, *MMU* надасть або заборонить цей доступ. У разі заборони доступу, ОС буде повідомлена. Коли виконання перемикається на процес *A*, інший набір таблиць сторінок буде завантажено в *MMU*. Загалом це може призвести до зовсім інших політик контролю доступу, оскільки таблиці сторінок різних процесів можуть зберігатися повністю незалежними. Таким чином, різні частини основної пам’яті можуть бути призначені виключно для різних процесів. Однак також можна спільно використовувати пам’ять

між програмами, відображаючи ту саму фізичну сторінку в різних адресних просторах або відображаючи її кілька разів під різними псевдонімами в одному адресному просторі.

### 2.1.2 Традиційні моделі загроз

У поєднанні обидва механізми (розмежування рівнів привілеїв та віртуальної пам'яті) дозволяють ОС забезпечити ізоляцію прикладного програмного забезпечення під час виконання. Моделі противника, які зазвичай розглядаються в цій ситуації, можна умовно розділити на два варіанти, які зображені на рис. 2.2.

Сьогодні багато атак використовують багатокрокові експлойти, в яких зловмисник послідовно мігрує від віддаленої до локальної моделі супротивника, щоб отримати повний контроль над платформою. Оскільки операційна система реалізує більшість методів зміцнення безпеки, атаки на програмне забезпечення ядра на цьому другому етапі стають все більш поширеними на практиці.



а) б)

Рисунок 2.2 – Варіанти атаки на операційну систему

На рис. 2.2 а) зображено віддаленого зловмисника, який атакує платформу ззовні, наприклад, через мережеве з’єднання. У цьому випадку довірена обчислювальна база системи включає всі програми користувача простору, для яких зловмисник може надати певну форму введення. Багато реальних сценаріїв можна

моделювати таким чином. Наприклад, зловмисник може мати доступ до спільного сховища файлів у локальній мереж або на цільовій систем може бути запущений веб-сервер, до якого зловмисник може підключитися через Інтернет. Якщо відповідна серверна програма мстить уразливість, зловмисник може використати це, щоб використати сервер і отримати деякий початковий контроль над комп’ютером, який атакований у віддаленому режим. У деяких випадках зловмисник може захотіти отримати інформацію або метою може бути зміна конфігурації веб-сервера. В інших випадках зловмисник може використати скомпрометований процес для атаки на інші частини системи або навіть зламати інші комп’ютери. На рис. 2.2 б) показана модель для локального зловмисника, яка припускає, що зловмисник уже контролює процес користувача на машині жертви. Це означає, що зловмисник може виконати довільний непривілейований код. Однак більшість сучасних комп’ютерних систем запускають сотні процесів паралельно, отже, операційні системи створені таким чином, щоб сприймати процеси користувача як ненадійні та потенційно шкідливі апріорі. З цією метою більшість основних ОС використовують різні методи, такі як обов’язковий контроль доступу [6], пісочниця [7,8], рандомізація макета адресного простору за замовчуванням. Аналіз показав, що атаки на ядро як наступний крок набувають дедалі більшої актуальності з наступних причин:

- ядро виконується з вищими привілеями, що часто дозволяє зловмисникові отримати повний контроль над машиною на основі одного ядра;
- операційна система реалізує основну частину методів посилення безпеки, згаданих вище. Отже, отримання кореневого процесу або вихід з пісочниці часто вимагає компрометації ядра.

Таким чином, багато атак у реальному світі включають багатоетапний підхід, у якому зловмисник починає з віддаленої атаки та послідовно переходить до локальної атаки, спочатку отримуючи виконання коду в просторі користувача, а потім атакуючи ОС. Отже ОС стала високоцінною мішенню, оскільки експлойти ядра використовуються в усіх найновіших *root*-експлойтах *Android* [9,10] джейлбрейках *iOS* [11], вихідних сигналах ізольованого програмного середовища, навіть широкомасштабні атаки на промислові системи управління.

## 2.2 Атаки з пошкодженням пам'яті

Атаки в реальному часі, що засновані на пошкодженні пам’яті, створюють ряд складних викликів для базових засобів захисту в рамках традиційних моделей загроз. Розглянемо основні причини програмного пошкодження пам’яті та визначимо, що є експлойтом її пошкодження.

### 2.2.1 Причини пошкодження пам'яті

Питання пошкодження пам’яті виявилось надзвичайно складним з точки зору безпеки. Частково це пов’язано з тим, що пошкодження пам’яті тісно пов’язане з способом визначення поведінки програми за допомогою мов низького рівня.

Відомо, що розробник програмного забезпечення закладає заплановану поведінку програми в код, а компілятор потім перекладає цей високорівневий зрозумілий людині опис у машинний код, який може бути виконано безпосередньо процесором. Однак, і ці мови явно допускають невизначену поведінку, тобто програми, які не дотримуються суворого набору технічних специфікацій, визначених у стандарті мови, можуть демонструвати будь яку поведінку під час виконання. Теоретично будь яка програма, написана сучасними мовами програмування, може містити вразливості, що призводять до пошкодження пам’яті.

Емпіричні дані за майже 40 років показують, що значна частина реального програмного забезпечення з часом схильна до вразливості, що пов'язана з пошкодженням пам'яті, і часто зустрічаються у розробленому коді [12]. Також, патчі, які виправляють ряд старих помилок, також можуть вводити нові помилки [13]. Дійсно, вразливості, що пов'язані з пошкодженням пам'яті, являють собою величезну кількість помилок, пов'язаних з безпекою сучасного програмного забезпечення, написаного на мовах низького рівня [14], які можуть бути використані зловмисниками, наприклад, для отримання віддаленого виконання коду або підвищених привілеїв під час виконання програми.

Вразливості пошкодження пам’яті можна класифікувати згідно з їх головним

дефектом. До типових прикладів вразливостей пошкодження пам’яті можна віднести: цілочисельні переповнення, некоректне використання комірки пам’яті після звільнення, висячі вказівники, подвійне звільнення комірки пам’яті, переповнення буфера, перевірка відсутніх вказівників, використання неініціалізованих даних, помилки типу або помилки синхронізації.

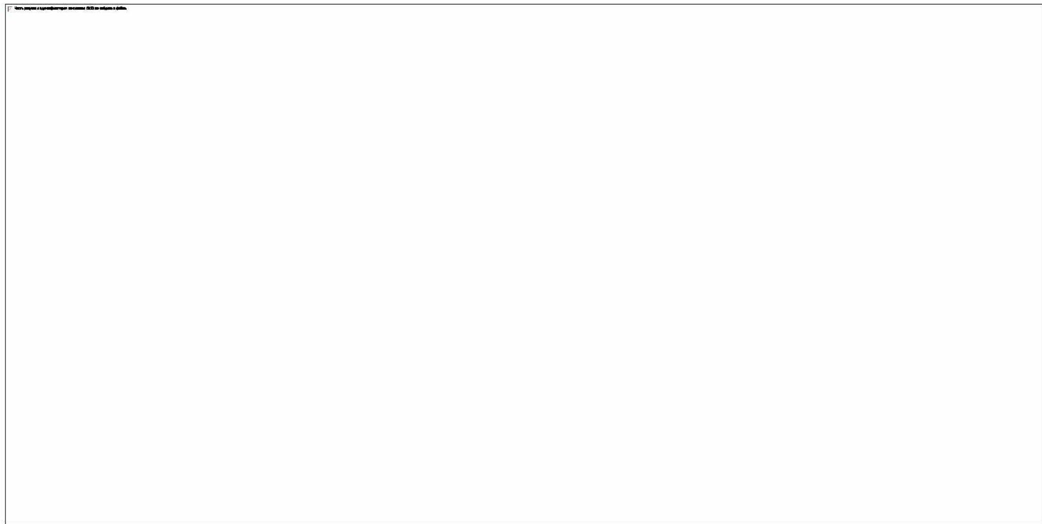
У стандарті *ANSI C* / *C*89 / *ISO C*90 вказано 96 випадків, які спричиняють невизначену поведінку і далі зазначається, що реалізація мови "може генерувати попередження у багатьох ситуаціях, жодна з яких не визначена як частина Стандарту" [15]. Це означає, що на практиці розробники програмного забезпечення можуть легко пропустити один з перелічених випадків у своєму коді, випадково написавши програму, яка буде демонструвати невизначену поведінку під час виконання. Одним з наслідків може бути пошкодження пам'яті, тобто ситуація, коли частина пам'яті програми, а отже, і загальний стан програми більше не є чітко визначеним. Простим прикладом такого випадку є переповнення буфера. У цій ситуації розмір деякого керованого користувачем входу повинен бути записаний у комірку пам'яті, але перевищує ємність буфера, зарезервованого у цій комірці. Це може бути використано зловмисниками для зловмисної модифікації даних або навіть перехоплення потоку керування програмою під час виконання.

### 2.2.2 Експлойт пошкодження пам’яті

Розглянемо виконання програми як послідовність переходів у машині зі скінченною кількістю станів (рис. 2.3). На рис 2.3 а) зображено стани передбачуваної поведінки машини зі скінченною кількістю станів, як спочатку вказав програміст. У цьому випадку існує виділений початковий стан (1) програми, з якого починається виконання. Переходи між станами вказуються розробником програмного забезпечення в коді, наприклад, у цьому випадку розробник передбачав, що програма нескінченно перемикається між трьома станами (2), (3) і (4) після виходу з початкового стану. На рис. 2.3 б) зображено, що у випадку невизначеної поведінки та сама програма може фактично демонструвати ненавмисні, приховані або стани, які

недоступні під час очікуваного виконання.

Якщо вразливість можливо активувати явно, зловмисник зможе перевести програму в приховані стани, які є вигідними з точки зору атаки.



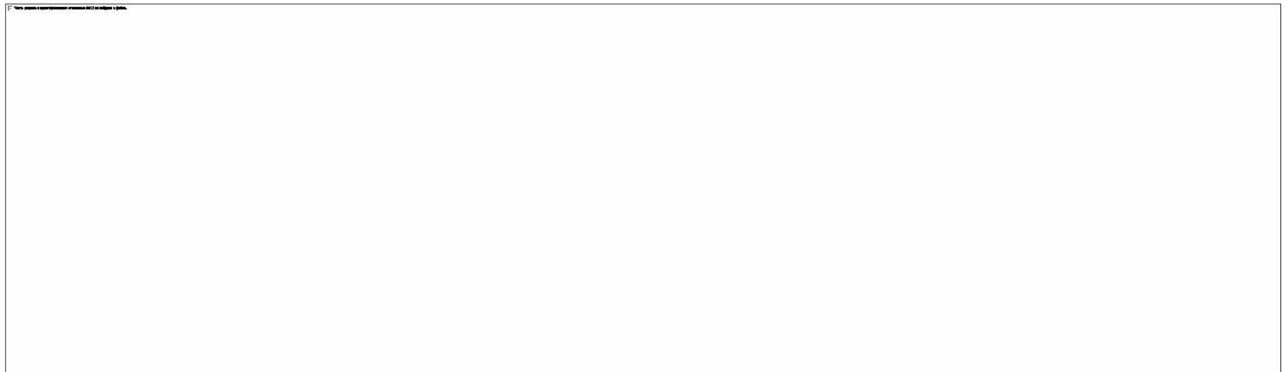
а) б)

Рисунок 2.3 – Послідовність переходів у машині зі скінченною кількістю станів

Отже, код, який розробник вказав для такої програми, стає абсолютно безглуздим, і призначена машина зі скінченною кількістю станів замінюється машиною з непередбачуваною поведінкою. Це також видно з рис. 2.3 б), на якому програма містить прихований кінцевий стан (6), на відміну від початково визначеної машини зі скінченною кількістю станів, який нескінченно перемикається між трьома визначеними станами. У цьому абстрактному погляді на використання фактичний експлойт представляє вхідний сигнал або тригер, який програмує машину на демонстрацію невизначеної або ненавмисної поведінки під час функціонування. Важливо відзначити, що ця абстрактна модель повністю підтримується специфікацією мови [16].

Розглянемо приклад робочого циклу експлойту пошкодження пам’яті, який наведено на рис. 2.4. На рисунку 2.4 а) програма використовує застарілу функцію, щоб зчитувати введені користувачем дані та зберігати їх у попередньо визначеному буферному сховищі без перевірки довжини введення. Хоч програміст і зарезервував пам’ять для 17 символів у коді програми, користувач може надати вхідні дані довільного розміру під час виконання. У більшості випадків це призведе до збою програми.

Однак зловмисник може надати спеціально створений вхід, подібний до рис. 2.4 б), який примусово переведе програму в непередбачений стан рис. 2.4 с), що породжує термінальну програму та надає зловмиснику інтерактивне середовище програмування. Це досягається шляхом перезапису адреси повернення головної функції, яка зберігається в стеку.



а) програма б) експлойт с) пошкоджений стек

Рисунок 2.4 – Експлойт пошкодження пам’яті

Будь-який випадок пошкодження пам’яті робить програму вразливою для атак під час виконання, а типові методи експлуатації варіюються від впровадження зловмисного коду до повторного використання існуючого коду зі зловмисним введенням і до пошкодження інтегральних структур даних запущеної програми без викрадення її потоку керування. Зловмисник, який знає про будь-яку таку вразливість, потенційно може використати це під час виконання, навмисно ініціюючи помилку для досягнення ненавмисної зловмисної поведінки та контролю над системою.

## 2.3 Сучасні атаки та захист

### 2.3.1 Захист від повторного використання коду.

Атаки під час виконання, засновані на вразливостях, пов’язаних з пошкодженням пам’яті, були постійною загрозою для прикладного програмного забезпечення та програмного забезпечення на рівні ядра протягом більше трьох

десятиліть [17]. Спочатку атакам з впровадженням коду приділялася велика увага, оскільки проста вразливість переповнення буфера дозволяла зловмисникам вводити довільний код у запущену програму, наприклад, надаючи зловмисні вхідні дані, які створюють оболонку. Атаки цього типу запобігають шляхом розгортання апаратних засобів захисту, таких як *Data Execution Prevention* (*DEP*) [18], які вимикають виконання пам’яті даних на платформі як загальну політику. Таким чином, зловмисники більше не можуть виконати код, який вони ввели в пам'ять програми. Однак базова парадигма атак була адаптована для обходу *DEP* як захисту шляхом узагальнення атак з впровадженням коду до повторного використання коду на тактах [19]. Під час атаки повторного використання коду зловмисник використовує вразливість до пошкодження пам’яті, щоб захопити потік керування запущеною програмою, зловмисно модифікуючи вказівник коду замість впровадження будь-якого коду, наприклад, адресу повернення, яка зазвичай зберігається в стеку програми можна змінити так, щоб вона вказувала на деяке місце довільного коду, яке вже є в пам’яті. У найпростішому випадку це дозволяє зловмиснику перенаправити виконання на іншу функцію, таку як функція бібліотеки, яка розгалужує іншу програму, надати зловмисний вхід (наприклад, «*/bin/sh*» для запуску оболонки). Прикладом цього є зворотно-орієнтоване програмування (*ROP*) [19], у якому виконання перенаправляється на невеликі фрагменти окремих інструкцій (так звані гаджети), які закінчуються інструкцією повернення. Оскільки зловмисник може з’єднувати довільні послідовності гаджетів разом, то можна вважати, що *ROP*-атаки є повними за Тьюрингом, це означає, що зловмисник може виконувати довільні обчислення, а отже, *ROP*-атаки є потужним інструментом зловмисника. Зазначимо, що атаки повторного використання коду можливі в багатьох системах, деякі з яких навіть не пропонують спеціальної інструкції повернення. Зокрема, можливості динамічного (або непрямого) розгалуження достатньо для створення атак повторного використання коду [20].

Аналіз сучасної літератури показав, що для захисту існуючих систем від таких атак доцільно використовувати методи на основі рандомізації або методи на основі політики захисту.

Методи на основі рандомізації спрямовані на зниження переносимості вразливості між ідентичними системами шляхом рандомізації частин віртуального адресного простору між програмами [21]. Ідея рандомізації макета адресного простору (*ASLR*) вперше була реалізована в 2000 році для рандомізації макета віртуальної пам'яті процесів користувача. Зокрема, метод *ASLR* рандомізує стек, купу та спільні бібліотеки перед завантаженням програми. Захист, заснований на рандомізаці, зазвичай має компроміс між безпекою та продуктивністю, де підвищена деталізація рандомізації призводить до більш високої ентропії, а також зазвичай створює додаткові витрати [22]. Окрім атак з вгадуванням методом грубої сили, які на практиці стають здійсненними через потребу понад 20–30 біт ентропії, захист на основ рандомізації зазнає атак з розкриттям інформації. Рандомізація розміщення адресного простору (*ASLR*) є добре відомою та широко використовуваною технікою захисту, яка рандомізує адреси пам’яті процесів, щоб запобігти формам експлуатації, які покладаються на знання точного розташування об’єктів процесу. Замість підвищення безпеки шляхом усунення вразливостей із системи, як зазвичай роблять інструменти аналізу вихідного коду, метод *ASLR* є профілактичною технікою, яка намагається ускладнити використання наявних уразливостей. На відміну від інших методів безпеки, безпека, яку забезпечує *ASLR*, базується на кількох факторах, включаючи те, наскільки передбачуваним є випадковий макет пам’яті програми, наскільки техніка експлуатації толерантна до варіацій у макеті пам’яті та кількості спроб, які зловмисник може зробити практично. Метод *ASLR* – це техніка захисту широкого спектру, у тому сенсі, що замість усунення особливого типу вразливості, як це робить *renewSSP*, вона ставить під загрозу програмний код зловмисників незалежно від вектора, який використовується для ін’єкції. Подібно до інших методів пом’якшення, метод *ASLR* пом’якшує атаки на виконання коду шляхом збою програми, і таким чином атака перетворюється на відмову в обслуговуванні. Метод *ASLR* – це абстрактна ідея, яка має кілька реалізацій, хоча між ними існують важливі відмінності в продуктивності та охопленні безпеки. Метод *ASLR* є ефективним захистом від сучасних атак.

Витік інформації розкриває інформацію про розташування пам'яті або окремі

адреси процесу, що дає зловмиснику можливість скоригувати експлойт. Прикладами використання вразливостей розкриття пам’яті є *Adobe Flash*, *PDF Reader* від *Adobe*, *Internet Explorer* від *Microsoft*. З цієї причини були запропоновані більш детальні методи рандомізації, наприклад, рандомізація на функціональному рівні [22] або на рівні інструкцій [23]. Єдиний витік коду *pointer* дозволяє зловмиснику повністю дерандомізувати програму під час виконання, розбираючи рандомізовані кодові сторінки, дотримуючись нещодавно виявлених покажчиків коду. Таким чином, зловмисник може динамічно збирати корисне навантаження ІКС за допомогою непрямого розкриття кодових сторінок. Хоча витоки інформації створюють значні проблеми для детальних схем рандомізаці, запропонована техніка захисту полягає в тому, щоб зробити кодові сторінки нечитабельними [24], бо це перешкоджає непрямим атакам на розкриття інформації. Однак стандартне апаратне забезпечення не підтримує нечитабельні кодові сторінки, отже, реалізація цього захисту потребує гіпервізора та підтримки розширеної таблиці сторінок. *Code-Pointer Integrity* [25] пропонує розділити пам'ять даних програми на область для покажчиків коду та область для решти даних. Рандомізуючи розташування області покажчика коду, можна вважати, що всі доступи до покажчиків коду безпечні. Визначено, що підхід, заснований на компіляторі, створює менш накладні витрати, ніж рандомізація захисту від атак повторного використання коду [26].

На відміну від цих імовірнісних методів захисту на основі рандомізаці, цілісність потоку керування (*Control Flow Integrity*, *CFI*) [27] виникла як метод захисту, який пропонує формальні гарантії безпеки, несучи незначні накладні витрати.

Цілісність потоку керування – це механізм безпеки, призначений для захисту комп'ютерних систем від атак на потік керування, таких як ін'єкції коду та атаки на повторне використання коду. Ці атаки передбачають, що зловмисник маніпулює потоком управління програми для виконання шкідливого коду або перехоплення виконання програми.

Основна мета *CFI* полягає в тому, щоб забезпечити дотримання потоком управління програми заздалегідь визначеного набору правил або графа потоку

управління. Він спрямований на запобігання відхиленням від запланованого потоку управління, тим самим ускладнюючи зловмисникам використання вразливостей і виконання довільного коду.

*CFI* забезпечує цілісність потоку управління за допомогою різних методів, зокрема:

- Граф потоку управління (CFG). Потік управління програми аналізується і представляється у вигляді графа, який фіксує всі можливі шляхи виконання. Цей граф потім використовується як еталон для перевірки потоку управління під час виконання.
- Точки забезпечення виконання потоку управління (CFEP). CFEP вставляються в певні місця коду програми. Ці точки виконують перевірки під час виконання, щоб переконатися, що потік керування відповідає CFG. Якщо виявлено порушення, спрацьовує виняток або попередження.
- Захист вказівників коду. CFI захищає вказівники коду, такі як вказівники функцій або адреси повернення, від маніпуляцій зловмисників. Він пов'язує метадані з кожним вказівником, перевіряючи їхню дійсність під час виконання та запобігаючи зловживанням.
- Тіньові стеки. CFI може використовувати тіньові стеки для зберігання додаткової інформації про потоки керування. Тіньовий стек зберігає окрему копію адрес повернення та інших даних, пов'язаних з потоком керування. Порівнюючи значення в тіньовому стеку з реальним стеком, метод *CFI* може виявляти і запобігати атакам на стек на основі потоку керування.

Метод *CFI* є ефективним механізмом захисту від атак на потік керування, оскільки він зосереджений на забезпеченні цілісності потоку керування програми. Завдяки суворому дотриманню заздалегідь визначеного графа потоку управління та захисту вказівників коду, зловмисникам стає значно складніше використовувати вразливості та виконувати шкідливий код.

Однак важливо зазначити, що хоча метод *CFI* забезпечує додатковий рівень захисту, він не є панацеєю і повинен використовуватися разом з іншими заходами безпеки, такими як перевірка вхідних даних, безпечні методи кодування та

пом'якшення наслідків використання уразливостей. Крім того, метод *CFI* може впливати на продуктивність через перевірки під час виконання та додаткові накладні витрати під час виконання.

Ідея високого рівня, що лежить в основі методу *CFI*, полягає в аналізі програмного коду під час компіляції для обчислення міток у всіх можливих розташуваннях розгалужень вставлення перевірок, які обмежують потік керування правильною міткою. Оскільки цей підхід вимагає виконання додаткових перевірок під час виконання для непрямих стрибків, початкові реалізації викликали накладні витрати від 5 до 20 відсотків, що все ще вважається непрактичним для багатьох програм [28]. Крім того, точність попередньо обчислених примусових наборів міток є важливою для досягнення високого рівня безпеки. Розгортання методу *CFI* в контекст об'єктно-орієнтованого коду виявилося складним, оскільки було показано, що реалізації, як нехтують детальною семантикою мов, таких як *C*++, піддаються атакам. Для двійкового коду неточні набори міток можуть призвести до атак, незважаючи на те, що застосовуються грубі політики методу *CFI*.

Таким чином, безпечна реалізація методу *CFI* зазвичай потребує поглибленого статичного аналізу та етапу інструментарію під час компіляції, дорогих перевірок під час виконання та реалізації тіньового стеку. Щоб зменшити втрату продуктивності, було досліджено реалізацію методу *CFI* з апаратним забезпеченням було оголошено про апаратні розширення, які пропонують спеціальну підтримку методу *CFI* для майбутніх архітектур ЦП.

### 2.3.2 Атаки лише на дані ядра ОС

Під атакою на дані ядра ОС будемо розуміти зміну даних ядра в пам'яті, зловмисником для маніпулюванням роботою операційних систем без ін'єкції шкідливого коду. Відомо, що ОС є фундаментальним блоком у стеку програмного забезпечення, що лежить в основі всіх програм користувача. З цією метою ядро абстрагує та керує складністю апаратного забезпечення реального часу та надає чітко визначені інтерфейси для процесів користувача. Крім того, ядро зазвичай працює з

підвищеними привілеями, а отже, є цінною мішенню для атак. Також майже всі ядра операційної системи реалізовані на небезпечних мовах програмування, що робить їх схильними до атак на основі пошкодження пам'яті. Оскільки основні ядра додатково пропонують розширену підтримку нових апаратних функцій, різноманітних конфігурацій і великої кількості пристроїв, вони також часто включають драйвери сторонніх виробників.

На відміну від атак впровадження коду та повторного використання коду, атаки лише на дані [29] не захоплюють потік керування додатком. Натомість зловмисник втручається у вхідні дані функцій або безпосередньо пошкоджує структури даних у пам'яті. Хоча потоки даних, як правило, сильно залежать від програми, раніше було показано, що атаки лише на дані серйозно впливають на безпеку вразливого програмного забезпечення [30]. Крім того, хоча атаки лише на дані не завжди можуть запропонувати таку ж гнучкість, як атаки з повторним використанням коду, вони можуть становити значну загрозу на практиці. Наприклад, зловмисник може змінити поточний ідентифікатор користувача, щоб підвищити привілеї, або змінити імена файлів, щоб розкрити секрети, так як конфігураційні файли або криптографічні ключі.

### 2.3.3 Способи захисту від атаки на дані ядра ОС

Аналіз сучасної технічної літератури показав, що для захисту від атак на дані ядра доцільно використовувати наступні методи: метод захисту виконання в режимі супервізора (*SMEP*), метод захисту доступу в режимі супервізора (*SMAP*), метод рандомізації розташування таблиць сторінок (*MPPTC*), які гарантують, що сторінки користувача не будуть доступні під час виконання ядра для захисту від атак.

Метод запобігання виконанню коду в режимі супервізора (*SMEP*) полягає в запобіганні виконанню коду, який розташований на користувацькій сторінці, за поточного рівня привілеїв, що дорівнює 0. З погляду атакуючого, цей засіб значно ускладнює експлуатацію вразливостей режиму ядра, тому що у цих умовах відсутнє місце для зберігання шелл-коду. Зазвичай під час експлуатації вразливості режиму

ядра атакуючий виділяє буфер із шелл-кодом у користувацькому режимі і потім активує вразливість, отримуючи контроль над виконанням коду і перевизначаючи точку виконання на вміст підготовленого буфера. Таким чином, якщо зловмисник не може виконати свій шелл-код, уся атака безглузда. є частиною механізму захисту сторінок пам'яті. Метод *SMEP* використовує вже існуючий прапор у записі таблиці сторінок - прапор *U/S* (прапор *User/Supervisor*, 2-й біт). Цей прапор вказує на те, є ця сторінка сторінкою користувацького режиму або режиму ядра. Власник сторінки визначає наявність доступу до даної сторінці пам'яті, тобто, якщо сторінка належить ядру операційної системи, яка виконує код у привілейованому режимі, доступ до неї з користувацького додатка неможливий. Під час спроби виконання коду, розташованого на користувацькій сторінці, у привілейованому режимі апаратно генерується помилка сторінки (*page fault*) внаслідок порушення прав доступу.

Запобігання доступу в режимі супервізора (*SMAP*) — це функція, реалізована в певних комп'ютерних процесорах для підвищення безпеки та захисту від певних типів уразливостей. В основному він призначений для запобігання несанкціонованому доступу до областей пам'яті в режимі супервізора, також відомому як режим ядра. У сучасних операційних системах існує два основних рівні привілеїв: режим користувача та режим супервізора (також відомий як режим ядра або привілейований режим). Режим супервізора має вищі привілеї та може виконувати важливі системні функції. Однак певні вразливості можуть дозволити зловмиснику отримати неавторизований доступ до областей пам'яті, зарезервованих для ядра операційної системи, що може призвести до атак на підвищення привілеїв і порушити безпеку системи. *SMAP* допомагає пом'якшити ці вразливості, запровадивши апаратний захист пам'яті. Це досягається шляхом позначення областей пам'яті як режиму лише супервізора, що означає, що будь-яка спроба отримати доступ до цих областей із режиму користувача призведе до апаратної виняткової ситуації. Застосовуючи це обмеження на апаратному рівні, *SMAP* забезпечує додатковий рівень захисту від певних типів атак на пам'ять. Коли програма працює в режимі користувача та намагається отримати доступ до області пам'яті, позначеної для доступу лише в режимі супервізора, процесор запускає помилку, що зазвичай

призводить до викликаного винятку або переривання. Цей механізм ефективно запобігає несанкціонованому доступу до критичних областей пам'яті ядра та допомагає захистити систему від експлойтів, які покладаються на такий доступ. *SMAP* є прикладом апаратної функції безпеки, яка доповнює програмні заходи безпеки, реалізовані операційними системами. Це може допомогти підвищити рівень безпеки систем і захистити від певних типів вразливостей, якими можуть скористатися зловмисники, щоб порушити цілісність системи та отримати неавторизовані привілеї. Варто зазначити, що хоча *SMAP* може бути ефективною функцією безпеки, це не ідеальний засіб і його слід використовувати разом з іншими методами безпеки та технологіями для створення надійного та безпечного системного середовища.

Інший метод запобіганню атакам лише на дані на таблиці сторінок у ядрі ОС є метод рандомізації розташування таблиць сторінок (*MPPTC*). Ідея цього методу полягає в рандомізації віртуальних адрес всіх сторінок таблиці у ядрі за допомогою загального рандомізованого зміщення. Забезпечується стійкість до витоків, зберігаючи це базове зміщення в привілейованому реєстрі та замінюючи всі покажчики даних на об'єкти таблиці сторінок їхніми фізичними адресами. Слід зазначити, що фізичні адреси не можуть використовуватися жодним програмним забезпеченням, якщо віртуальну пам'ять увімкнено ОС під час завантаження. Це означає, що всі доступи до посилань на таблиці сторінок у *MPPTC* можна отримати лише за допомогою секрету рандомізації. Це ключовий момент для захисту *MPPTC* від атак лише на дані - зловмисник не отримує жодних знань від розкриття фізичних покажчиків таблиці сторінок. Зокрема, зловмисник не може використовувати їх для читання таблиць сторінок під час перегляду сторінок або змінювати необхідний запис таблиці сторінок. Для перетворення посилань на фізичні таблиці сторінок у придатні для використання віртуальні адреси потрібен секрет рандомізації, який надійно зберігається в реєстрі.

## Висновок до другого розділу

На теперішній час, для захисту операційних систем використовують розмежування рівнів привілеїв та захист віртуальної пам’яті. Визначено, що служби операційних систем стандартизовані у формі системних викликів і найчастіше використовуються в якості основи атаки на користувацьке програмне забезпечення. Показано, що різні частини основної пам’яті призначені виключно для різних процесів, але можна спільно використовувати пам'ять між програмами, відображаючи ту саму фізичну сторінку в різних адресних просторах або кілька разів під різними псевдонімами в одному адресному просторі.

Дослідження показало, що сучасні атаки використовують багатокрокові експлойти, в яких зловмисник послідовно мігрує від віддаленої до локальної моделі порушника, щоб отримати повний контроль над ІКС.

Атаки в реальному часі, що засновані на пошкодженні пам’яті створюють ряд складних викликів для базових засобів захисту в рамках традиційних моделей загроз.

Аналіз існуючих методів захисту ІКС від повторного використання коду показав, що доцільно використовувати метод, який заснований на рандомізації макета адресного простору, а також метод цілісності потоку керування.

Визначено, що іншою поширеною загрозою інформаційної безпеки ІКС є атаки на дані ядра ОС. Дослідження показало, що для захисту від такого типу атаки доцільно використовувати метод захисту виконання в режимі супервізора (*SMEP*), метод захисту доступу в режимі супервізора (*SMAP*), метод рандомізації розташування таблиць сторінок (*MPPTC*), які гарантують, що сторінки користувача не будуть доступні під час виконання ядра для захисту від атак.

# РОЗДІЛ 3
# ОЦІНКА МЕТОДІВ ЗАХИСТУ ПРОГРАМНИХ КОМПЛЕКСІВ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНОЇ СИСТЕМИ

Під час налаштування захищених систем важливо враховувати гарантії, які можуть бути гарантовані механізмами забезпечення безпеки. У розділі 2 було визначено основні типи атак на ОС. Уразливість до пошкодження пам’яті зазвичай спричинена відсутністю перевірки вхідних даних у створеній програмі, і програмістам надається свобода визначати, коли та як обробляти вхідні дані. Ця гнучкість часто призводить до кращої продуктивності програми. Прикладами атак з пошкодженням пам’яті є переповнення буфера, переповнення купи та переповнення стека. Атаки переповнення буфера є одним із найпоширеніших класів експлойтів проти цілісності потоку керування комп’ютерними системами. Ця загроза ще більше ускладнюється загальною залежністю організацій від стороннього та закритого програмного забезпечення. Процес виявлення та виправлення таких вразливостей у вихідному коді не тільки буде несвоєчасним і дорогим, але й вихідний код не завжди доступний для виправлення. Основна мета цієї атаки – скористатися переповненням буфера для отримання привілейованого контролю над програмою. В даному розділі детально проведено аналіз сучасних методів захисту від атак переповнення буфера

## 3.1 Метод рандомізації розміщення адресного простору (ASLR)

Рандомізація адресного простору (*Address Space Layout Randomization, ASLR*) - це технологія, яка використовується для запобігання атакам і ґрунтується на знанні розташування цільового коду або даних. Ефективність *ASLR* базується на тому, що вся структура адресного простору залишається невідомою зловмиснику. Максимальний захист від техніки *ASLR* можуть отримати лише виконувані файли, скомпільовані як *Position Independent Executable* (*PIE*), оскільки всі секції завантажуються у випадкових місцях. Це захисний механізм, який рандомізує адреси

пам'яті процесів, намагаючись запобігти спробам експлуатації. Замість того, щоб підвищувати безпеку шляхом видалення вразливостей з системи, як це роблять інструменти аналізу вихідного коду [31], *ASLR* є профілактичною технікою, яка намагається ускладнити використання існуючих вразливостей і сумісне використання методів *Stack Smashing Protector* (*SSP*), *No-eXecute* (*NX*) забезпечить надійний захист від більшості атак, що пов’язані з пошкодженням пам’яті.

Концепція *ASLR*: всі адреси, які зловмисники можуть використовувати для побудови експлойтів, повинні бути їм невідомі (або важко вгадувані), але повна реалізація (всі області розташовані у випадкових місцях) може мати проблеми з сумісністю; також ще одним обмеженням є обмежений діапазон адрес для виділення областей, більшість 32-бітних систем мають лише 8 біт ефективної ентропії.

Безпека, яку пропонує *ASLR*, базується на декількох факторах, включаючи те, наскільки непередбачуваною є випадкова структура пам'яті програми та скільки спроб нападу може здійснити зловмисник на практиці.

Основна ідея *ASLR* полягає в тому, щоб розмістити всі області процесу (дані, купу, текст, бібліотеки, тощо) за випадковими адресами. Випадкові адреси можна визначити як адреси, які невідомі і важко вгадуються зловмисниками. Випадковість адресного простору перешкоджає деяким типам атак, оскільки зловмиснику складніше передбачити цільові адреси. Ці значення потрібно вгадати (або обчислити), щоб успішно обійти *ASLR*. Безпека *ASLR* базується на низькій ймовірності зловмисника вгадати розташування випадково розміщених областей, а отже, чим більша ентропія, тим безпечнішою є система. На рис. 3.1 представлено роботу методу *ASLR*. Коли процес запускається, операційна система випадково розподіляє адреси компонентів пам'яті, включаючи виконуваний код, бібліотеки, стек і купу. Адреси компонентів пам'яті рандомізуються шляхом додавання випадкового зсуву до базових адрес. Це означає, що кожного разу під час запуску процесу адреси будуть різними. Всередині процесу використовуються відносні адреси, а не абсолютні. Програма звертається до відносних адрес, і операційна система переводить їх у відповідні абсолютні адреси.

Існує три різні виміри ентропії для кожної області:

1. Невипадкова: загальновизнано, що навіть незначна нерандомізована область може бути використана зловмисником для подолання *ASLR*. Тому всі області повинні бути рандомізовані.

2. Діапазон ентропії: розмір або діапазон можливих значень, в якому може знаходитись кожна область. Чим більший діапазон, тим краще.

3. Частота переміщення: кожен процес повинен мати власний простір пам'яті, де всі області розташовані в різних місцях відносно попередніх екземплярів того ж самого виконуваного файлу та відносно інших паралельних процесів. На більшості систем початкові реалізації *ASLR* покладалися на інфраструктуру спільної бібліотеки. Тому *ASLR* спочатку застосовувався лише до бібліотек, що було дуже ефективно проти прямих атак типу *return-2-lib*. Розвиток *ROP* (*Return Oriented Programming*) [32] та пов'язаних з ним технік дозволив зловмисникам будувати експлойти практично на будь-якій ділянці коду, яка не була рандомізована, що стимулювало потребу в повноцінній реалізації *ASLR*.

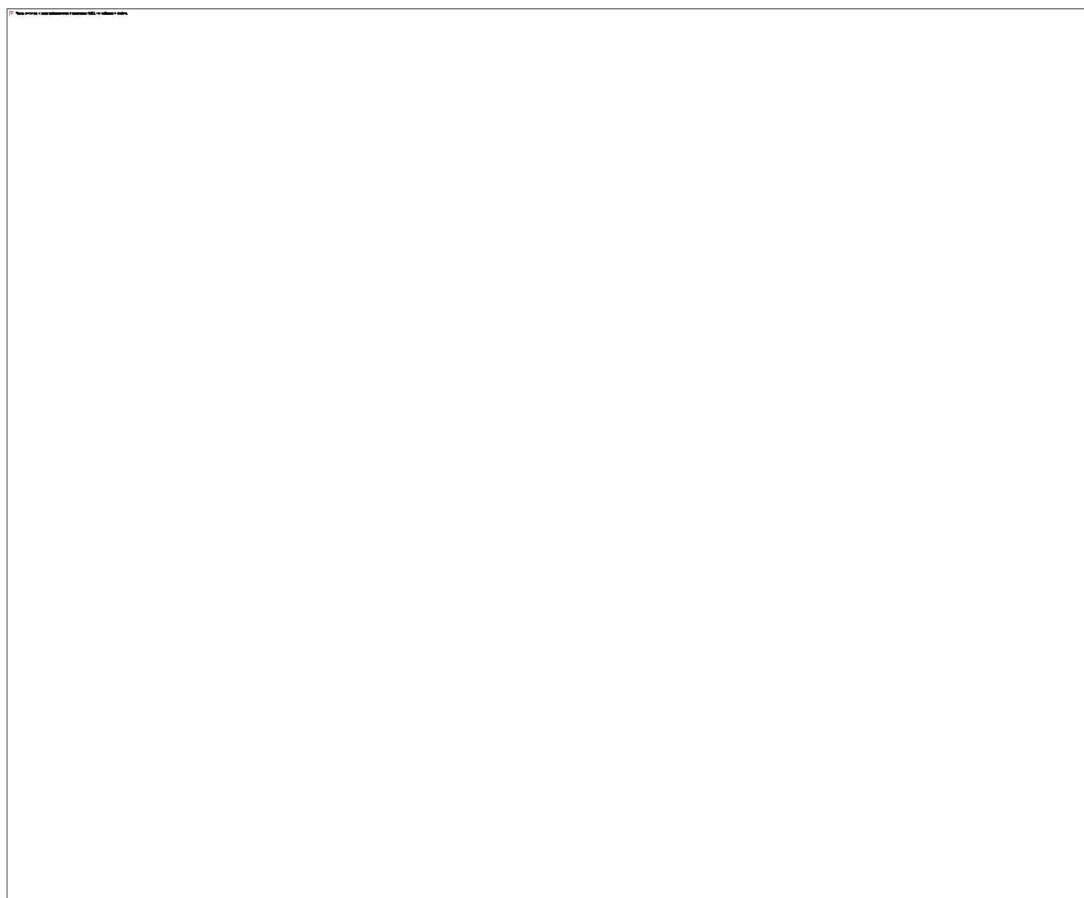


Рисунок 3.1 – Вплив методу *ASLR* на розміщення компонентів в пам’яті

Діапазон ентропії серйозно обмежений доступним об'ємом віртуальної пам'яті. Практично неможливо мати "пристойну" реалізацію на 32-бітних системах; з 256 можливими значеннями вона вважається майже марною. Проста атака перебором може зламати *ASLR* за кілька мілісекунд. Але на 64-бітних системах цей діапазон досить великий, щоб ефективно протистояти зловмисникам, якщо не існує іншого методу вилучення інформації з цільового процесу. Навіть у нереалістичних атаках, де система не забезпечує бітовий захист *SSP* та *NX*, час, необхідний для обходу *ASLR*, коливається між 1,7 годинами та 34,1 годинами. Останнє джерело ентропії - це частота оновлення, яка безпосередньо пов'язана з тим, як обробляються та використовуються спільні бібліотеки. Якщо спільні бібліотеки мають бути відображені на однакові віртуальні адреси у всіх процесах, то *ASLR* може бути виконано лише на основі "завантаження". Тобто, лише під час першого завантаження бібліотеки вона буде випадковим чином зіставлена. Наступні процеси повинні використовувати бібліотеку у вже відображеному місці. Ця послідовність створює єдине відображення бібліотек у пам'яті на системному рівні, яке оновлюється лише при перезавантаженні системи.

До переваг методу *ASLR* відноситься:

1. Метод *ASLR* ускладнює атаки, які намагаються використати вразливості, випадковим чином розміщуючи важливі компоненти, такі як стек, купа, бібліотеки *DLL* і функції, в адресному просторі пам'яті. Це ускладнює зловмисникам визначення правильних адрес для виконання шкідливого коду або захоплення потоку керування, що ускладнює успішне використання.

2. Метод *ASLR* зменшує вплив уразливості: навіть якщо вразливість виявлено, *ASLR* ускладнює для зловмисників пошук правильних адрес для виконання шкідливого коду або викрадення потоку керування. Це допомагає обмежити вплив уразливості та зменшує ризик успішної атаки.

3. Метод *ASLR* може захищати від певних типів атак, таких як атаки на покажчик функції (перехоплення функцій) або зворотно-орієнтоване програмування. Випадкове розміщення компонентів ускладнює побудову правильних послідовностей покажчиків або пошук необхідних функцій для створення ланцюжків *ROP*.

До недоліків методу *ASLR* можна віднести:

1. Передбачуваність: використання рандомізації може мати певний ступінь передбачуваності, особливо в деяких випадках із застарілими версіями операційної системи або програмами, які не мають достатньої рандомізації. Досвідчені зловмисники зможуть обійти *ASLR*, якщо точно передбачать шаблони рандомізації.

2. Уразливості впровадження: ефективність *ASLR* значною мірою залежить від його правильного впровадження. Якщо в реалізації є вразливі місця або слабкі місця, зловмисники можуть скористатися ними та порушити захист від рандомізації.

*ASLR* створює певні накладні витрати на продуктивність через необхідність додаткових звернень до пам'яті та трансляцій адрес. Процес рандомізації та наступні пошуки адрес можуть призвести до дещо сповільненого виконання програм, хоча зазвичай вплив мінімальний.

## 3.2 Метод цілісності потоку керування (CFI)

Цілісність потоку керування – це політика захисту, яка обмежує ненавмисну передачу потоку керування в несанкціоновані місця [33]. Метод *CFI* дозволяє захищатися від різних типів атак, основною метою яких є перенаправлення потоку керування в інше місце. Аналіз наукової літератури показав, що за останні кілька років було запропоновано багато методів *CFI*. Однак вони не були повністю прийняті через практичні проблеми та значні обмеження. Розглянемо основні методи цілісності потоку керування. Проведене дослідження дозволило класифікувати методи цілісності потоку керування на *Fine-Grained CFI* та *Coarse-Grained CFI*. Обидві класифікації містять переваги та недоліки з точки зору безпеки потоку керування.

3.2.1 Класифікація методів *CFI*.

Метод *Fine-Grained CFI* також називається *CFI* строгого типу [34]. Початок та кінець передачі потоку керування містять мітки, і їх цілісність перевіряється під час кожної передачі керування. Перед передачею потоку керування процес перевірки гарантує, що передача передається до дійсного пункту призначення, який містить дійсну мітку. На рис. 3.2 показано приклад обходу методу *Fine-Grained CFI*.

Зловмисник може успішно перенаправити (стрілка “*Attack*”) потік керування з інструкції *ret* на кроці 4 на інструкцію *mov* після виклику *f*1(). Потік виконання складається з прямих і непрямих викликів. Прямий виклик є фіксованим і не може бути зміненим, тоді як цільова адреса обчислюється перед поверненням непрямого виклику. Непряме повернення 4 є дійсним і також повертає до дійсного пункту призначення. Однак атака ініціюється в тій же точці повернення, перезаписуючи адресу повернення, і зловмиснику вдається перенаправити потік в інше місце. Метод *Fine-Grained CFI* зазвичай застосовує тіньовий стек, який викликає інтенсивні накладні витрати [35].

Отже, незважаючи на те, що метод *Fine-Grained CFI* підвищує надійність безпеки йому не вистачає продуктивності і водночас він є дорогим. Тому була запропонована інша форма *CFI* – *Coarse-Grained CFI.*

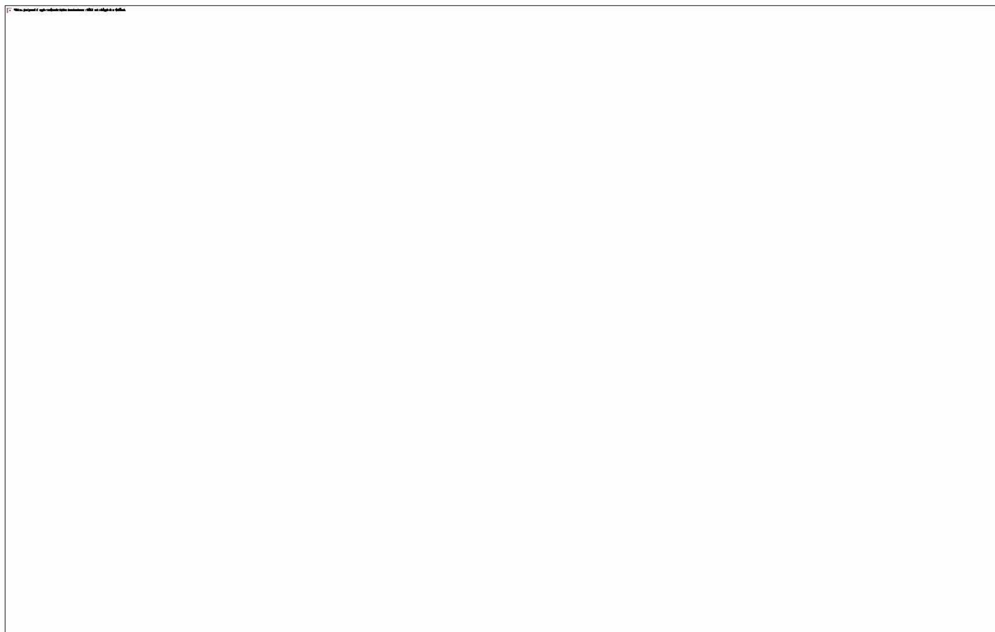


Рисунок 3.2 – Приклад обходу методу *Fine-Grained CFI*

Метод *Coarse-Grained CFI* розрізняє непрямі інструкції розгалуження за їхньою класифікацією та накладає політику на кожен їх тип [34]. *Coarse-Grained CFI* перевіряє, коли передача потоку керування починається з інструкції повернення, щоб адресат міг бути визначений одразу після інструкції виклику. Цей підхід *CFI* є більш м’яким і забезпечує бажану продуктивність, але доведено, що він небезпечний [35].

На рис. 3.3 показано приклад обходу методу *Coarse-Grained CFI.* Потік

керування складається з прямих і непрямих переходів потоку керування. Як прямі, так і непрямі інструкції виклику/повторення від 1 до 4 є дійсними та підтвердженими *Coarse-Grained CFI*. Однак непрямий виклик, *call *eax*, є вказівником, отже, окрім переходу до дійсної функції призначення *f*2(), зловмисники також можуть перенаправити потік за допомогою незаконних переходів до функції *f1*() та ініціювати атаку потоку керування. Зловмисник може перенаправити потік керування з кількох викликів і точок відновлення. Стрілки “*Attack*” показують дійсні пункти призначення, куди зловмисники можуть перенаправити потік. Дослідження показало, що метод *Coarse-Grained CFI* не є повним захистом від атак керування потоком, тоді як метод *Fine-Grained CFI* вважається більш безпечним.

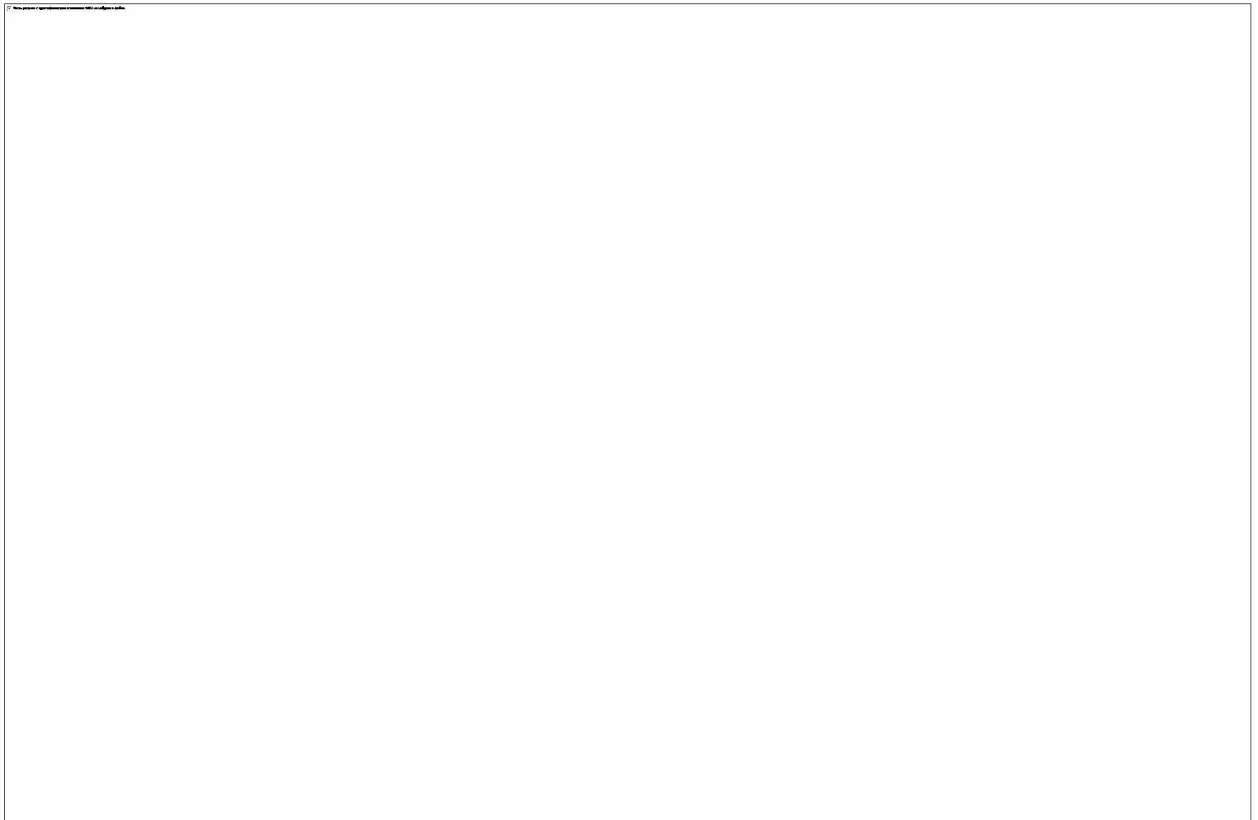


Рисунок 3.3 – Приклад обходу методу *Coarse-Grained CFI*

### 3.2.2 Варіації методів CFI, для запобігання перенаправлення потоку керування.

У [36] запропоновано *CFI*1, що реалізований для *Windows* на платформі ×86. Ініціюється граф потоку керування (*CFG*) перед виконанням програми. *CFG* відстежує поведінку під час виконання, отже, будь-яка невідповідність у програмі призводить до виклику виняткової ситуації *CFI* та завершення програми. Інструмент *CFI* не впливає на прямі виклики функцій, але лише непрямі виклики вимагають перевірки ідентифікатора. Функції, які викликаються опосередковано, наприклад віртуальні методи, потребують вставки ідентифікатора. Ця техніка *CFI* гарантує, що керування програмою передається правильному адресату. Таким чином, щоразу, коли машинний код передає своє керування, призначення має бути дійсним, а також попередньо визначеним *CFG*. Даний метод має наступні обмеження:

1) вихідний код не завжди доступний;
2) у двійкових файлах не вистачає необхідної налагоджувальної інформації;
3) спричиняє великі витрати на виконання через динамічне перезаписування та перевірки під час виконання;
4) не в змозі впоратися з динамічним кодом або двійковими файлами, які суперечать конвенціям компілятора [37];
5) не може визначити, чи функція повертається до поточного виклику.

Метод *Cryptographically Enforced CFI* (*CCFI*) має нові механізми вказівників (міток), які не можуть бути нав'язані статичними підходами [38]. Складається з двох основних атрибутів:

1) перекласифікує вказівники на функції під час виконання, щоб покращити приведення типів;
2) обмежує заміну двох дійсних вказівників одного типу.

Метод *CCFI* включає динамічний механізм *CFI*, який використовує криптографічні примітиви. Це детальне удосконалення компілятора, у якому коди автентифікації повідомлень (*MAC*) реалізовано на кожному вказівнику потоку керування та накладає різні атрибути часу виконання та мови. Метод *CCFI*

розроблено для захисту адрес повернення, вказівника кадру, вказівників на функції, вказівників *vtable* та обробників винятків. Ключ *MAC* створює випадкові значення на початку програми, які зберігаються в регістрах. Випадкові зсуви додаються до кожного кадру *malloc* і стеку, щоб рандомізувати кожну пам'ять і кадр стеку. Метод *CCFI* є вразливим до атак відтворення [34], а також не в змозі ідентифікувати вказівники структури, і може порушити потік керування, змінивши поточний вказівник на старий. Метод *CCFI* в основному зосереджується на захисті програми на рівні користувача і не включає безпеку на рівні ядра [38]. Функції безпеки не зосереджені на захисті адрес купи та стека.

Метод *CFI* для бінарних файлів *COTS* (*BinCFI*) використовується для підвищення безпеки вразливих додатків, зменшення можливості зламу контролю потоку виконання програми та запобігання атакам, які використовують перехоплення і зміну потоку виконання. Основна ідея *BinCFI* полягає в тому, що бінарні файли модифікуються з метою внесення інструкцій та додаткових даних для перевірки цілісності контролю потоку. Ці модифікації дозволяють виявити некоректні зміни потоку виконання та забезпечити його безпечну траєкторію. Основні компоненти методу *BinCFI* включають:

- інструментування бінарних файлів. Бінарний файл зазначеного COTS-продукту модифікується, додаванням інструкцій та метаданих, які допомагають виявляти та перевіряти контроль потоку виконання;
- механізм перевірки цілісності. Внесені зміни дозволяють механізму перевіряти, чи відбувається потік виконання за очікуваною траєкторією, а також чи відбуваються неправомірні зміни контролю потоку. Це може включати перевірку адрес переходів, викликів функцій або стеку викликів;
- захистні заходи. Якщо механізм перевірки цілісності виявляє некоректні зміни контролю потоку, він може приймати заходи для запобігання атакам. Це може включати відхилення некоректних переходів, виведення повідомлень про помилки або припинення виконання програми.

Метод *BinCFI* зазвичай вимагає модифікації самого бінарного коду, тому він може бути складним для застосування до комерційних продуктів. Однак, він може

забезпечити значні покращення безпеки, зменшуючи ризик атак, пов'язаних з контролем потоку виконання.

Метод практики контролю потоку виконання (*CFI*) та рандомізації для двійкових виконуваних файлів (*CCFIR*). Практика контролю потоку виконання (*CFI*) та рандомізації для двійкових виконуваних файлів (*CCFIR* - *Control-Flow Integrity and Randomization*) поєднує дві важливі техніки безпеки для захисту від атак, пов'язаних з контролем потоку виконання програм. *CFIR* розроблений з метою захисту від атак, що спрямовані на злам контролю потоку виконання, таких як ретельно сплановані атаки переповнення буфера, інъєкція коду, розгалуження віддаленого коду (*ROP* - *Return-Oriented Programming*) і *JOP* (*Jump-Oriented Programming*). Основна ідея *CCFIR* полягає в поєднанні двох підходів:

1) контроль потоку виконання. Це включає механізми, що перевіряють, чи потік виконання програми відповідає заданій траєкторії. Це може бути досягнуто за допомогою додаткових метаданих, перевірки хеш-сум контрольних точок або використанням аналізу потоку даних. Ці перевірки забезпечують, що програма виконується відповідно до очікуваної структури керування потоком;

2) рандомізація. Це включає зміну розташування коду та даних в пам'яті під час кожного запуску програми. Застосування рандомізації ускладнює завдання атакуючих, оскільки вони не можуть передбачити точне розташування коду або адреси функцій, необхідних для виконання зловмисного коду. Рандомізація може включати зміну адресного простору виконуваних файлів, стеку викликів, розташування функцій та інших компонентів програми.

Комбінація *CFI* і рандомізації у *CCFIR* допомагає забезпечити високий рівень захисту від різних атак, які використовуються для злому контролю потоку виконання. Ці методи можуть застосовуватися як окремо, так і разом, для забезпечення більш ефективного та надійного захисту від атак.

Метод *HW-CF*I запропоновано *NSA Information Assurance*. Відмінною особливістю даного методу є впровадження *CFG* на апаратне забезпечення та захист динамічного контрольного потоку, що захищений тіньовим стеком. Адреса повернення виклику функції буде збережена як в області стека, так і в тіньовому

стеку. Під час повернення до користувача обидві адреси будуть зіставлені для перевірки. Будь-яка відмінність призведе до збою ЦП і призведе до завершення програми. Оскільки помилки програмного забезпечення можуть пошкодити дані та призвести до можливих загроз, примітним аспектом цього методу *CFI* є те, що дані в тіньовому стеку будуть захищені апаратними функціями. Метод *HW-CFI* може бути не сумісним з усіма системними архітектурами, також явний моніторинг тіньового стека часто неможливий.

Метод *PICFI* накладає обчислену *CFG* на кожен вхід. Звичайно, важко розглянути всі входи програми та *CFG* для кожного входу. Таким чином, *PICFI* запускає програму з порожнім *CFG* і дозволяє програмі самостійно знайти *CFG* [39]. Основна ідея цього порожнього підходу *CFG* полягає в тому, щоб прикріпити мітки до середовища виконання перед використанням для інструкцій розгалуження. *PICFI* обмежує додавання міток, які не були визначені під час статичного обчислення *CFG*. Метод *PICFI* містить способи оптимізації продуктивності. Метод *PICFI* статично обчислює *CFG*, щоб визначити мітки, які будуть додані під час виконання, і реалізує *Data Execution Prevention* для захисту від впровадження коду.

Метод *KCoFI* забезпечує захист стандартних операційних систем від атак, таких як модифікація сегмента коду. Метод *KCoFI* включає звичайний підхід на основі міток для роботи з непрямими інструкціями розгалуження. Метод *KCoFI* не базується на повному аналізі програм чи складному загальному захисті пам'яті. Метод *KCoFI* відповідає всім вимогам управління обробкою подій, але результат цього примусового виконання *CFI* є занадто дорогим [40].

Метод *Kernel CFI* повністю реалізує підхід до модернізації *FreeBSD*, системи мікроядра *MINIX*, сервера простору користувача *MINIX* і частково гіпервізора *BitVisor* [40]. Цей метод *CFI* використовує два основні підходи до реалізації *CFI*. По-перше, вихідний код ядра містить шаблони використання вказівників на функції, які можна використовувати для створення техніки, яка допомагає обчислювати *CFG* для програмного забезпечення ядра. По-друге, виконання двох методів для виключення інструментів *CFI*, пов'язаних із попередньою технікою оптимізації. Хоча метод *Kernel CFI* скорочує непряму передачу до 70%, але існує шанс для передачі непрямих

інструкцій розгалуження до непередбаченого пункту призначення.

Метод *Forward-Edge CFI* в *GCC* & *LLVM* (*IFCC*) — це детальна реалізація *CFI*, яка передбачає аналіз для компілятора *GNU Compiler Collection* (*GCC*) і *Low-Level Virtual Machine* (*LLVM*). Непрямі перевірки викликів функцій (*IFCC*) — це механізм перетворення *CFI*, який реалізується через компілятор *LLVM* 3.4. Метод *IFCC* представляє динамічний інструмент, який можна використовувати для аналізу та визначення вразливостей *CFI* [41]. Метод направлений на:

- захист даних, які зберігаються в купі;
- аналіз під час виконання, який розроблений для виявлення потоку керування.

Метод *IFCC* не в змозі захистити від контрольної атаки *Jujutsu*, яка спрямована на виконання зловмисного корисного навантаження [42]. Іншим основним обмеженням цього механізму *CFI* є те, що він не може перевірити, коли функція повертається назад до місця призначення [43].

Метод *CFB* містить статичну реалізацію *CFI* [44]. Метод *CFB* може бути прив'язаний до атаки, яка встановлює передачу потоку керування в законному *CFG*. Метод *CFB* реалізує повністю точну статичну *CFG*, яка може бути нерозв'язною [42]. Статичний аналіз *CFI* містить значні недоліки, оскільки вони не можуть визначити кожне можливе значення вказівника функції. *CFB* також порушує певні функції на високому рівні, і виконання таких функцій може змінити адресу повернення та пошкодити потік керування.

Метод *SAFEDISPATCH* забезпечує захист від викрадення *vtable*. Він статично перевіряє програми *C++* і виконує перевірку під час виконання, щоб переконатися, що потік керування на сайтах викликів віртуальних методів не захоплений зловмисниками [45]. Метод *SAFEDISPATCH* — це вдосконалений компілятор *C++*, створений на основі *Clang++/LLVM*. Метод *SAFEDISPATCH* не може захистити двійкові файли, що робить його вразливим до атак на основі *ROP*. Оскільки різні програми часто використовують спільну бібліотеку або динамічно завантажувані бібліотеки, усі детальні ілюстрації на основі компілятора зазнають загальних проблем безпеки, пов'язаних із спільно використовуваними бібліотеками.

Метод *Control Flow Guard* — це механізм безпеки, що розроблений *Microsoft*

для захисту від різних типів уразливостей пам'яті. Він захищає від використання помилок пам'яті шляхом посилення безпеки там, де програма виконує свої коди. Виконання довільного коду досить важко ініціювати через поширені вразливості, такі як переповнення буфера. Метод *C-Guard* не може перевірити, коли функція повертається до несанкціонованого місця призначення.

Метод захисту від повторного використання (*RAP*) запобігає зворотно-орієнтованому програмуванню, реалізованому як плагін компілятора *GCC*. *RAP* використовує ядро *Linux* під час компіляції для реалізації *CFI* під час виконання та гарантує, що вказівники коду не пошкоджуються зловмисником. Однак він не має надійного захисту від атак, таких як *ret2usr* [46]. Реалізований підхід *RAP* дуже схожий на традиційний підхід на основі міток. Експерименти показують, що версія *RAP* з відкритим вихідним кодом може бути успішно використана.

Метод непрозорого *CFI* (*O-CFI*) містить бінарну програмну рандомізацію для примусового виконання *CFI* [35]. Він захищає застарілі двійкові файли навіть без доступу до вихідного коду. Метод *O-CFI* здатний приховати графік потоку керування від зловмисників, які отримають повний доступ до перегляду стека, купи та іншої інформації про програму. Метод *O-CFI* передбачає статичний бінарний підхід, тому він не може захистити код, який генерується динамічно. Він несумісний із моделлю компонентних об'єктів *Windows* (*COM*). Метод також не в змозі вирішити проблеми, пов'язані з впровадженням *CFI* для *COTS* сучасних програм *Windows* [37].

Кожен метод захисту *CFI* може бути реалізований на апаратному та програмному забезпеченні, використовувати різні підходи, такі як тіньовий стек, мітки, тощо. В табл. 3.1 наведено ключові характеристики, які застосовуються проаналізованими методами захисту *CFI*.

Показано, що всі методи захисту *CFI* засновані на програмному забезпеченні, і лише три з них також реалізовані апаратно. Крім того, більшість із них потребують модифікації компіляторів і виконують офлайн-графік потоку керування (*CFG*) програми.

Методи *CFI* є надійним механізмом захисту від викрадення потоку керування. Незалежно від апаратного чи програмного забезпечення, обидва підходи базуються

на подібних методах безпеки. Наприклад, статичний і динамічний підхід із тіньовим стеком використовується для більшості методів *CFI*.

*Таблиця 3.1*

Основні характеристики методів захисту CFI

| *CFI Technique* | *Based on* | | *Compiler Modified* | *Shadow Slack* | *CFC* | *label* | *Coarse Grained* | *Fine Grained* | *Backward Edge* | *CFI Enforcement* |
|---|---|---|---|---|---|---|---|---|---|---|
| | *HW* | *SW* | | | | | | | | |
| *CFI1* | | ✓ | | ✓ | ✓ | ✓ | ✓ | | ✓ | *Inlined CFI* |
| *CCFI* | ✓ | ✓ | ✓ | | | | | ✓ | ✓ | *Dynamic Analysis* |
| *binCFI* | | ✓ | | | | ✓ | ✓ | | | *Static Binary Rewriting* |
| *CCFIR* | | ✓ | | | | ✓ | ✓ | | ✓ | *Binary Rewriting* |
| *HWCFI* | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | *Landing Point* |
| *PICFI* | | ✓ | ✓ | | ✓ | | | ✓ | ✓ | *Static Analysis* |
| *KCoFL* | | ✓ | ✓ | | | ✓ | ✓ | | ✓ | *SVA Instrumentation* |
| *Kernel CFI* | | ✓ | | | ✓ | | | ✓ | ✓ | *Retrofitting Approach* |
| *IFCC* | | ✓ | ✓ | | ✓ | | | ✓ | | *Dynamic Analysis* |
| *CFB* | | ✓ | | ✓ | ✓ | | | ✓ | ✓ | *Precise Static CFI* |
| *SAFE DISPATCH* | | ✓ | ✓ | | | | ✓ | | | *Static Analysis* |
| *C-Guard* | | ✓ | ✓ | | ✓ | | ✓ | | | *Dynamic Instrumentation* |
| *RAP* | | ✓ | ✓ | | ✓ | | | ✓ | ✓ | *Type Based* |
| *O-CFI* | ✓ | ✓ | | | | | ✓ | ✓ | ✓ | *Static Rewriting* |

Статичний аналіз відбувається перед виконанням, на етапі запису об’єктного коду, не в змозі перевірити, чи вказівник функції вказує на певний пункт призначення під час виконання. Крім того, динамічний аналіз спостерігає за станом виконання програми [47], однак він часто створює хибнопозитивні та хибнонегативні результати. Більшість методів *CFI* на основі програмного забезпечення залежать від динамічного інструментарію, де вихідний код вставляється в програму для проведення оцінки *CFI* для непрямих інструкцій розгалуження. З іншого боку,

тіньовий стек був прийнятий основними методами *CFI*. Його реалізація покращує здатність захищати зворотний край і має апаратну підтримку *Intel* [48]. Однак тіньовий стек страждає від серйозних проблем із сумісністю.

### 3.3 Порівняння рівня безпеки між методами цілісності потоку керування і рандомізацією адресного простору для захисту від атак переповнення буфера в операційній системі Windows 10

У [49] проведено дослідження з метою порівняння методів *ASLR* і *CFI* в програмах на базі *Windows* 10.

Дане дослідження проводилось експериментальним методом. Кожен метод (*CFI* та *ASLR*) призначений для захисту програми під час компіляції вихідного коду. Кожен результат компіляції буде перевірено на переповнення стека та купи. Крім того, дослідження здійснювалось на основі методів атаки керування потоком, атак повторного використання коду та атак із впровадженням коду, та проаналізовано їх вплив на пам'ять за допомогою інструмента *Immunity Debugger*.

Методика оцінки здійснюються шляхом аналізу процесу компіляції вихідного коду та його подальшого запуску в *Windows* 10, а також аналізом відмінностей якості захисту, отриманих після компіляції вихідного коду.

Об'єкт дослідження використовує вихідний код додатку, який буде скомпільовано з використанням обох методів захисту (*ASLR* та *CFI*).

У підготовленому вихідному коді використано кілька функцій програми відповідно до контрольного списку платформи перевірки векторної атаки, де кожне меню представляє контрольний список.

Результати тестування представлені у вигляді табл. 3.2, вони ілюструють порівняння кількості атак, яким можна запобігти.

За результатами проведеного досліджень можна зробити наступні висновки [49]:

- рівень безпеки методу цілісності керування потоком кращий у обробці атак переповнення буфера в операційній системі *Windows* 10 порівняно з методом рандомізації розміщення адресного простору;

*Таблиця 3.2*

Результати порівняння методів

| *STACK OVERFLOW TO TARGET* | *ASLR* | *CFI* | *WITHOUT PROTECTION* |
|---|---|---|---|
| *Return value* | - | + | - |
| *base pointer* | - | + | - |
| *Pointer local variabel* | - | + | - |
| *Pointer function parameter* | - | + | - |
| *longjmp buffer* | *N/A* | *N/A* | *N/A* |
| | | | |
| *HEAP/BSS OVERFLOW TO TARGET* | | | |
| *Return value* | + | + | - |
| *base pointer* | + | + | - |
| *Pointer local variabel* | - | - | - |
| *Pointer function parameter* | - | - | - |
| *longjmp buffer* | *N/A* | *N/A* | *N/A* |

- метод рандомізації розміщення адресного простору є швидшим у часі виконання, ніж метод цілісності потоку керування (має подвійне збільшення часу виконання порівняно з *ASLR*).

**Висновок до третього розділу**

Аналіз методу рандомізації розміщення адресного простору дозволив визначити його переваги та недоліки. Показано, що метод *ASLR* ускладнює атаки випадковим чином розміщуючи важливі компоненти, такі як стек, купа, бібліотеки *DLL* і функції, в адресному просторі, а також ускладнює для зловмисників пошук правильних адрес для виконання шкідливого коду або викрадення потоку керування.

Проведено класифікацію та розглянуто основні методи *CFI*. Кожен з розглянутих методів має обмеження застосування та можуть бути зруйновані різними векторами атак. Визначено, що методи на основі програмного забезпечення є більш

безпечними порівняно з методами на основі апаратного забезпечення. Їх легше реалізувати і вони не вимагають змін архітектури ОС. На основі аналізу кожного методу *CFI* було визначено, що більшість методів *CFI* не функціональні для забезпечення належної безпеки, і, як наслідок, вони не були повністю освоєні виробниками ОС, а отже, немає жодного конкретного методу, який би забезпечував повний захист.

Проведено порівняння рівня безпеки між методами цілісності потоку керування і рандомізацією адресного простору для захисту від атак переповнення буфера в операційній системі *Windows* 10. Показано, що рівень безпеки методу цілісності керування потоком  кращий у обробці атак переповнення буфера в операційній системі *Windows* 10 порівняно з методом рандомізації розміщення адресного простору, але метод *ASLR* є швидшим у часі виконання, ніж метод *CFI*.

# ВИСНОВКИ

У кваліфікаційній роботі розв’язано актуальне завдання, яке полягає в аналізі сучасних атак та методів захисту операційної системи. В ході розв’язання поставлених задач були отримані наступні наукові та практичні результати:

1. Проведений аналіз функціонування існуючої ІКС дозволив представити обладнання, що використовується, у вигляді складного апаратно-програмного комплексу, що складається з двох взаємопов’язаних підсистем: апаратної і програмної. Програмний комплекс є важливою складовою частиною обладнання, що має обмежену інформаційну безпеку і суттєво впливає на безпеку ІКС в цілому.

2. Дослідження дозволило визначити, що операційна система є цінною мішенню для здійснення атак. Атаки в реальному часі, що засновані на пошкодженні пам’яті створюють ряд складних викликів для базових засобів захисту. Значна частина реального програмного забезпечення схильна до вразливостей, що пов’язані з пошкодженням пам’яті і часто зустрічається в розробленому коді. Переповнення буфера є одним з найбільш поширених викликів, що пов’язані з пошкодженням пам’яті.

3. Оцінка методів захисту програмних комплексів інформаційно-комунікаційної системи показало, що на теперішній час доцільно використовувати в якості захисту від атак з пошкодженням пам’яті методи *ASLR* та *CFI*. Метод *ASLR* пом’якшує атаки на виконання коду шляхом збою програми і таким чином атака перетворюється на відмову в обслуговуванні. На відміну від імовірнісних методів захисту на основі рандомізації метод *CFI* пропонує формальні гарантії безпеки та забезпечує дотримання потоком управління програми заздалегідь визначеного набору правил або графа потоку управління.

4. Виконана практична реалізація методів *ASLR* та *CFI*, що в подальшому дозволить проводити дослідження, що пов’язані з підвищенням захисту програмних комплексів інформаційно-комунікаційної системи.

Всі завдання поставленні під час виконання кваліфікаційної роботи виконано в повному обсязі.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Закон України “Про захист інформації в інформаційно-комунікаційних системах” [Електронний ресурс]. – Режим доступу:https://zakon.rada.gov.ua/laws/show/80/94-%D0%B2%D1%80#Text

2. Бєльков Д.В. Концепція мультисервісних мереж [Електронний ресурс]. – Режим доступу:http://ea.donntu.edu.ua/bitstream/

3. Дмитро Могилевич, Вікторія Сінько Моделі надійності об’єктів телекомунікаційного обладнання з незнеціненими або повністю знеціненими відмовами програмних засобів [Електронний ресурс]. – Режим доступу:http://its.iszzi.kpi.ua/article/download/261132/257589

4. Donald C Latham Department of defense trusted computer system evaluation criteria [Електронний ресурс]. – Режим доступу: https://nsarchive.gwu.edu/document/18466-national-security-archive-department-defense .

5. IEEE Computer Society - Austin Joint Working Group. 1003.1-2008 – IEEE Standard for Information Technology – Portable Operating System Interface (POSIX(R)). [Електронний ресурс]. – Режим доступу:https://ieeexplore.ieee.org/document/4694976

6. National Security Agency. Security-Enhanced Linux (SELinux). [Електронний ресурс]. – Режим доступу:https://www.nsa.gov/portals/75/documents/what-we-do/research/selinux/documentation/presentations/2005-flexible-support-for-security-policies-into-linux-os-presentation.pdf

7. Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. “A Secure Environment for Untrusted Helper Applications.” In: 6th USENIX Security Symposium. USENIX Sec. 1996. [Електронний ресурс]. – Режим доступу: https://www.researchgate.net/publication/2401297_A_Secure_Environment_for_Untrusted_Helper_Applications_Confining_the_Wily_Hacker

8. Jonathan Corbet. Yet another new approach to seccomp. [Електронний ресурс]. – Режим доступу: https://lwn.net/Articles/475043/

9. Nick Stephens. Behind the PWN of a TrustZone. [Електронний ресурс]. – Режим доступу: https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someoneunlock-your-phone-with-nose

10. Tencent. Defeating Samsung KNOX with Zero Privilege. [Електронний ресурс]. – Режим доступу: https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege-wp.pdf

11. Tarjei Mandt. Attacking the iOS Kernel: A Look at "evasi0n". [Електронний ресурс]. – Режим доступу: https://docplayer.net/61240561-Attacking-the-ios-kernel-a-look-at-evasi0n-tarjei.html

12. Jidong Xiao, Hai Huang, and Haining Wang. “Kernel Data Attack Is a Realistic Security Threat.” In: International Conference on Security and Privacy in Communication Systems. Springer. 2022 [Електронний ресурс]. – Режим доступу:https://link.springer.com/chapter/10.1007/978-3-319-28865-9_8

13. Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. “Memory errors: the past, the present, and the future.” In: International Workshop on Recent Advances in Intrusion Detection. Springer. 2023 [Електронний ресурс]. – Режим доступу: https://www.researchgate.net/publication/236278779_Memory_Errors_The_Past_the_Present_and_the_Future

14. Steve Christey and Robert A Martin. “Vulnerability type distributions in CVE.” In: Mitre report, May (2019). [Електронний ресурс]. – Режим доступу:https://cwe.mitre.org/documents/vuln-trends/index.html

15. ANSI Committee X3J11. ANSI C / C89 / ISO C90. [Електронний ресурс]. – Режим доступу: http://port70.net/~nsz/c/c89/c89-draft.html

16. Thomas F Dullien. “Weird machines, exploitability, and provable unexploitability.” In: IEEE Transactions on Emerging Topics in Computing. [Електронний ресурс]. – Режим доступу: http://www.dullien.net/thomas/weird-machines-exploitability.pdf

17. Ralf Hund, Thorsten Holz, and Felix C Freiling. “Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms.” In: USENIX Security Symposium. 2019. [Електронний ресурс]. – Режим доступу: https://www.usenix.org/legacy/events/sec09/tech/full_papers/hund.pdf

18. Microsoft. Data Execution Prevention (DEP). [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/memory/dataexecutionprevention

19. Hovav Shacham. “The geometry of innocent flesh on the bone: return-into libc without function calls (on the x86).” In: ACM SIGSAC Conference on Computer and Communications Security. CCS. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/1315245.1315313

20. Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. “Jump-oriented programming: a new class of code-reuse attack.” In: 6th ACM Symposium on Information, Computer and Communications Security. ASIACCS. [Електронний ресурс]. – Режим доступу:https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf

21. Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. “SoK: Automated Software Diversity.” In: 35th IEEE Symposium on Security and Privacy. S&P. [Електронний ресурс]. – Режим доступу: https://oaklandsok.github.io/papers/larsen2014.pdf

22. Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. “Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software.” In: 22nd Annual Computer Security Applications Conference. ACSAC. [Електронний ресурс]. – Режим доступу: https://ieeexplore.ieee.org/document/4041179

23. Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. “Binary stirring: self-randomizing instruction addresses of legacy x86 binary code.” In: ACM SIGSAC Conference on Computer and Communications Security. CCS. 2018. [Електронний ресурс]. – Режим доступу:https://personal.utdallas.edu/~hamlen/wartell12ccs.pdf

24. Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. “You Can Run but You Can’t Read: Preventing Disclosure Exploits in Executable Code.” In: ACM SIGSAC Conference on Computer and Communications Security. CCS. 2020. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/2660267.2660378

25. Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. “Code-Pointer Integrity.” In: 11th USENIX Sympo sium on Operating Systems Design and Implementation. OSDI. 2022. [Електронний ресурс]. – Режим доступу:https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-kuznetsov.pdf

26. Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. “Missing the Point(er): On the Effectiveness of Code Pointer Integrity.” In: 36th IEEE Symposium on Security and Privacy. S&P.. [Електронний ресурс]. – Режим доступу: https://ieeexplore.ieee.org/document/7163060

27. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-flow integrity.” In: ACM SIGSAC Conference on Computer and Communications Security. CCS. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/1102120.1102165

28. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-flow integrity principles, implementations, and applications.” In: ACM Transactions on Information System Security 13. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/1609956.1609960

29. Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. “Non-Control-Data Attacks Are Realistic Threats.” In: 14th USENIX Security Symposium. USENIX Sec. [Електронний ресурс]. – Режим доступу: https://www.usenix.org/legacy/publications/library/proceedings/sec05/tech/full_papers/chen/chen_html/index.html.bak.html

30. Hong Hu, Shweta Shinde, Adrian Sendroiu, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. “Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks.” In: 37th IEEE Symposium on Security and Privacy. S&P. 2016 [Електронний ресурс]. – Режим доступу: https://n.ethz.ch/~sshivaji/publications/dop_oakland16.pdf

31. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the Effectiveness of Address-Space Randomization,” in Proceedings of the 11th ACM Conference

on Computer and Communications Security, pp. 298–307, ACM, 2004. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/1030083.1030124

32. Shenglin Xu, Yongjun Wang Defending against Return-Oriented Programming attacks based on return instruction using static analysis and binary patch techniques [Електронний ресурс]. – Режим доступу: https://www.sciencedirect.com/science/article/abs/pii/S0167642322000016

33. Pappas, V. Defending against Return-Oriented Programming. 2015. [Електронний ресурс]. – Режим доступу: https://www.cs.columbia.edu/~angelos/Papers/theses/vpappas_thesis.pdf

34. Muench, M.; Pagani, F.; Yan, S.; Kruegel, C.; Vigna, G.; Balzarotti, D. Taming transactions: Towards hardware-assisted control flow integrity using transactional memory. In Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses, Paris, France, 19–21 September 2016. [Електронний ресурс]. – Режим доступу:https://link.springer.com/chapter/10.1007/978-3-319-45719-2_2

35. Mohan, V.; Larsen, P.; Brunthaler, S.; Hamlen, K.W.; Franz, M. Opaque Control Flow Integrity. In Proceedings of the 22nd Annual Network and Distributed System Security Symposium, San Diego, CA, US. [Електронний ресурс]. – Режим доступу: https://www.researchgate.net/publication/275971047_Opaque_Control-Flow_Integrity

36. Abadi, M.; Budiu, M.; Erlingsson, U.; Ligatti, J. Control-flow Integrity. The 12th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 7–11 November 2005; pp. 340–353. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/1102120.1102165

37. Hawkins, B.; Demsky, B.; Taylor, M.B. BlackBox: Lightweight Security Monitoring for COTS Binaries. In Proceedings of the 2016 International Symposium on Code Generation and Optimization, Barcelona, Spain, 12–18 March 2016; pp. 261–272. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/2854038.2854062

38. Mashtizadeh, A.J.; Bittau, A.; Boneh, D.; Mazières, D. CCFI: Cryptographically Enforced Control Flow Integrity. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA. pp. 941–951. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/2810103.2813676

39. Niu, B.; Tan, G. Per-Input Control-Flow Integrity. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA. pp. 914–926. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145 /2810103. 2813644

40. Ge, X.; Talele, N.; Payer, M.; Jaeger, T. Fine-Grained Control-Flow Integrity for Kernel Software. In Proceedings of the IEEE European Symposium on Security and Privacy, Saarbrucken, Germany, 21–24 March 2016; pp. 179–194 [Електронний ресурс]. – Режим доступу: https://nebelwelt.net/files/16EUROSP.pdf

41. Tice, C.; Roeder, T.; Collingbourne, P.; Checkoway, S.; Erlingsson, U.; Lozano, L.; Pike, G. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In Proceedings of the 23rd USENIX Conference on Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 941–955. [Електронний ресурс]. – Режим доступу: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf.

42. Evans, I.; Long, F.; Otgonbaatar, U.; Shrobe, H.; Rinard, M.; Okhravi, H.; Sidiroglou-Douskos, S. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, pp. 901–913. [Електронний ресурс]. – Режим доступу: https://people.csail.mit.edu/stelios/papers/ucs5.pdf.

43. Grsecurity. How Does RAP Works. [Електронний ресурс]. – Режим доступу: https://grsecurity.net/rap_faq.

44. Carlini, N.; Barresi, A.; Payer, M.; Wagner, D.; Gross, T.R. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In Proceedings of the 24th USENIX Conference on Security Symposium, Washington, DC, USA, 12–14 August 2015; pp. 161–176. [Електронний ресурс]. – Режим доступу: https://www.usenix.org/system/files/conference /usenixsecurity15 sec15-paper-carlini.pdf.

45. Jang, D.; Tatlock, Z.; Lerner, S. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks; Internet Society: San Diego, CA, USA,

2014. [Електронний ресурс]. – Режим доступу:https://www.researchgate.net/publication/269196977_SAFEDISPATCH_Securing_C_Virtual_Calls_from_Memory_Corruption_Attacks.

46. Li, J.; Tong, X.; Zhang, F.; Ma, J. FINE -CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. IEEE Trans. Inf. Forensics Secur. 2018, 13, 1535–1550. [Електронний ресурс]. – Режим доступу: https://ieeexplore.ieee.org/document/8269390

47. Zhang, M.; Qiao, R.; Hasabnis, N.; Sekar, R. A Platform for Secure Static Binary Instrumentation. SIGPLAN Not. 2014, 49, 129–140. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/2674025.2576208

48. Dang, T.H.; Maniatis, P.; Wagner, D. The Performance Cost of Shadow Stacks and Stack Canaries. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, Singapore, 14–17 April 2015; pp. 555–566. [Електронний ресурс]. – Режим доступу: https://dl.acm.org/doi/10.1145/2714576.2714635.

49. Avip Kurniawan, Nezim, Ridwan Comparison of Security Level between Control Flow Integrity (CFI) Method and Address Space Layout Randomization (ASLR) on Buffer Overflow Attack Protection on Windows 10 Operating Systems [Електронний ресурс]. – Режим доступу: http://www.ijctjournal.org/Volume5/Issue6/IJCT-V5I6P19.pdf

# ДОДАТОК А

## Список опублікованих праць за темою кваліфікаційної роботи


1. С. Даков, В. Могилевич, К. Малишев. Використання BLOCKCHAIN для збереження цілісності даних. МНПК *Проблеми кібербезпеки інформаційно-телекомунікаційних систем* (PCSITS). Київ: КНУ ім. Тараса Шевченка, 2022. – С. 144–145.

2. B. Kredentser, D. Mogylevych, I. Kononova, V. Mohylevych. Analytical Model with Interruption of Service of Short-term Objects with Temporary Reservation. *IT&I 2020. Information Technology and Interactions*. 2020 – P. 295 – 307. URL: https://ceur-ws.org/Vol-2845/Paper_28.pdf. (Scopus).

3. B. Kredentser, D. Mogylevych, I. Kononova, V. Mohylevych. Analytical Model with Interruption of Service of Short-term Objects with Temporary Reservation Information Technology and Interactions. *International conference Information Technology and Interactions* (Satellite). 2020 – P. 301 – 303. URL: http://iti.fit.univ.kiev.ua/wp-content/uploads/ITI_Satellite_2020_Conference-Proceedings.pdf (Scopus).

4. D. Mogylevych, I. Subach, B. Kredentser, I. Kononova, V. Mohylevych and V. Riabtsev, Optimization of the Frequency of Maintenance of Long-Term Storage Facilities with Taking into Account Time Reservation. *2022 IEEE 4th International Conference on Advanced Trends in Information Theory (ATIT)*, Kyiv, Ukraine, 2022, pp. 238-243, doi: 10.1109/ATIT58178.2022.10024202.

5. S. Dakov, V. Mohylevych, I. Parkhomenko. Using Blockchain For Data Storage. *Information Technology and Implementation (IT&Is-2022)*, Kyiv, Ukraine, 2022, pp. 35-36.

6. Д. Могилевич, І. Кононова, О. Климович, В. Могилевич. Методика комплексної оцінки надійності телекомунікаційного обладнання мереж зв’язку. Військово-технічний збірник, (23). 2020. – С. 50–57. DOI: 10.33577/2312-4458.23.2020.50-57.

7. В. Сінько, В. Могилевич. Аналіз факторів які впливають на надійність програмного забезпечення інформаційних систем спеціального призначення. XIII НПК

*Пріоритетні напрямки розвитку телекомунікаційних систем та мереж спеціального призначення. Застосування підрозділів, комплексів, засобів зв'язку, автоматизації та кібербезпеки в операції Об'єднаних сил*, Київ: ВІТІ ім. Героїв Крут, 2020. – С. 248

8. В. Могилевич. Аналіз атаки повторного використання коду на операційні системи. МНПК *Кіберборотьба: розвідка, захист та протидія.* Київ: ВІТІ ім. Героїв Крут, 2023. – С. 38.

# ДОДАТОК Б
## Лістинг програми

Метод *ASLR* є технікою безпеки, яка ускладнює експлуатацію вразливостей програм шляхом випадкового розміщення виконуваного коду, бібліотек та даних у віртуальному адресному просторі процесу. Розроблена програма на мові *Python*, яка дозволяє практично продемонструвати роботу методу *ASLR* за допомогою прапора *ADDR_RANDOMIZE* у *Linux*-системах. Функція *randomize_address* генерує випадковий зсув у межах 32-бітного адресного простору та застосовує його до вихідної адреси. Таким чином ми отримуємо випадкову адресу з урахуванням *ASLR*. Код використовує специфічну функцію *personality* із бібліотеки *libc* для встановлення прапора захисту процесу, тому він буде працювати лише на *Linux* системах.

```
import random
import ctypes
def enable_aslr():
    # Отримуємо поточні прапори захисту процесу
    current_flags = ctypes.CDLL('libc.so.6').personality(0xffffffff)
    # Включаємо прапор ADDR_RANDOMIZE, щоб увімкнути ASLR
    ADDR_NO_RANDOMIZE = 0x0040000
    new_flags = current_flags | ADDR_NO_RANDOMIZE
    ctypes.CDLL('libc.so.6').personality(new_flags)
def randomize_address(address):
    # Генеруємо випадковий зсув в межах 32-бітного адресного простору
    shift = random.randint(0, 0xFFFFFFFF)
    # Застосовуємо зсув до адреси
    randomized_address = address + shift
    return randomized_address
# Приклад використання
enable_aslr()
# Довільна адреса у віртуальному адресному просторі
address = 0x400000
# Отримуємо випадкову адресу з використанням ASLR
randomized_address = randomize_address(address)
print(f' Початкова адреса: 0x{address:X}')
print(f'Випадкова адреса з ASLR: 0x{randomized_address:X}')
```

На рис. Б.1 показано роботу програми.

Рисунок Б.1 - Робота програми

*Control Flow Integrity* (*CFI*) – це техніка безпеки, яка забезпечує перевірку правильності викликів функцій та переходів між ними. *CFI* зазвичай реалізується лише на рівні компілятора чи з допомогою інструментації коду. Розроблено спрощену програмну реалізацію *CFI* на *Python*, яка може демонструвати основні принципи роботи цього методу.

```
import functools
class ControlFlowIntegrity:
    def __init__(self):
        self.call_graph = {}  # Граф викликів функцій
    def add_function(self, function_name):
        # Додаємо функцію у граф викликів
        if function_name not in self.call_graph:
            self.call_graph[function_name] = []
    def add_call(self, caller_function, callee_function):
        # Додаємо зв'язок між функцією, що викликає, і функцією, що викликається
        if caller_function in self.call_graph and callee_function in self.call_graph:
            self.call_graph[caller_function].append(callee_function)
    def validate_call(self, caller_function, callee_function):
        # Перевіряємо правильність виклику функції
        if caller_function in self.call_graph and callee_function in self.call_graph[caller_function]:
            return True
        else:
            return False
    def validate_call_decorator(self, callee_function):
        def decorator(func):
            @functools.wraps(func)
            def wrapper(*args, **kwargs):
                caller_function = func.__name__
                if self.validate_call(caller_function, callee_function):
                    return func(*args, **kwargs)
                else:
                    raise ValueError("Invalid function call")
            return wrapper
```

```
    return decorator
cfi = ControlFlowIntegrity()
# Додаємо функції у граф викликів
cfi.add_function("function1")
cfi.add_function("function2")
cfi.add_function("function3")
# Встановлюємо дозволені виклики
cfi.add_call("function1", "function2")
cfi.add_call("function2", "function3")
@cfi.validate_call_decorator("function2")
def function1():
   print("Function 1")
@cfi.validate_call_decorator("function3")
def function2():
   print("Function 2")
def function3():
   print("Function 3")
# Викликаємо функції
function1()
function2()
function3()
```

На рис Б.2 показано роботу програми.

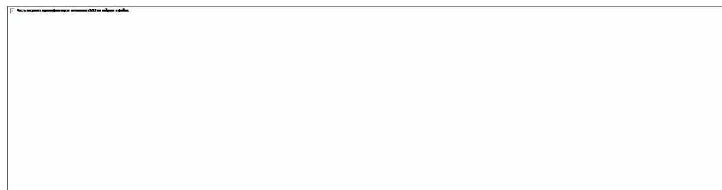


Рисунок Б.2 – Робота програми

У цьому прикладі *ControlFlowIntegrity* є класом, який моделює перевірку потоку управління програми. Можна використати декоратор *validate_call_decorator* для перевірки правильності функцій виклику. Декоратор приймає ім'я функції, що викликається, і перевіряє його на основі заздалегідь визначених дозволених викликів у графі викликів *call_graph*. В даному випадку, виклики функцій *function*1(), *function*2() та *function*3() проходять перевірку відповідно до дозволених викликів, визначених у графі викликів *call_graph*. Це означає, що всі три функції будуть успішно викликані та виведуть відповідні повідомлення. Якщо програма, що реалізує *CFI* на *Python*, виявить неприпустимий виклик функцій, вона може вивести

виключення *ValueError* з повідомленням *Invalid function call*.

Наприклад, якщо змінити код в такий спосіб:

```
@cfi.validate_call_decorator("function2")
def function1():
   print("Function 1")
@cfi.validate_call_decorator("function4")  # Неіснуюча функція
def function2():
   print("Function 2")
def function3():
   print("Function 3")
```

Тепер виклик *function*2() не відповідає дозволеним викликам, оскільки немає зв'язку між *function*1 та *function*2 у графі викликів. При виконанні такого коду буде згенеровано виключення *ValueError* з повідомленням *Invalid function call*. Таким чином, якщо програма виявить неприпустимий виклик функцій, вона може вивести повідомлення про помилку та припинити виконання програми (рис. Б.3).

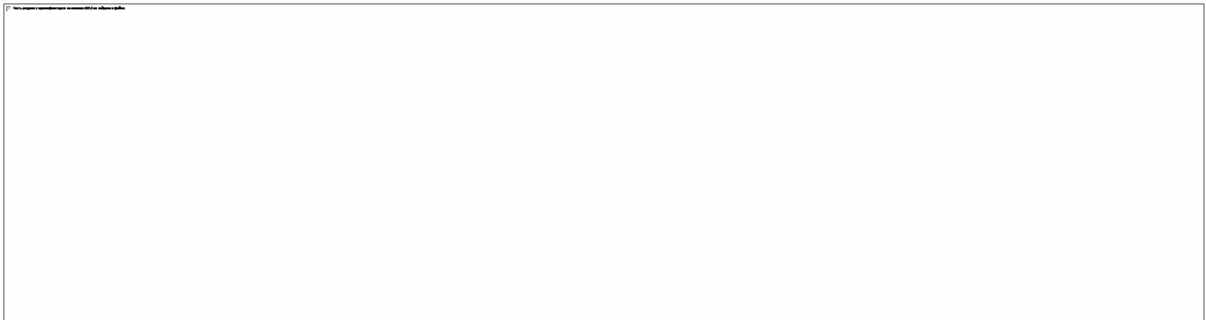


Рисунок Б.3 – Помилка при неправильному виклику