Міністерство освіти і науки України
Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп’ютерної інженерії та кібербезпеки

ПОЯСНЮВАЛЬНА ЗАПИСКА

до випускної кваліфікаційної роботи бакалавра

на тему: «Проєкт системи оцінювання вразливостей APK-файлів»

Виконав: студент 4-го курсу групи КБ-22-2
спеціальності 125 «Кібербезпека»
_____ Іван АРСЕНТЬЄВ

Керівник: асистент кафедри комп’ютерної інженерії
та кібербезпеки
_____ Артем ЛАЙЧУК

Рецензент: професор кафедри комп’ютерної інженерії
та кібербезпеки, доктор технічних наук, доцент
_____ Володимир ВОРОТНІКОВ

Житомир – 2026

Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки
Спеціальність 125 «Кібербезпека»
Освітня програма «Кібербезпека»

ЗАТВЕРДЖУЮ
Завідувач
кафедри комп'ютерної
інженерії та кібербезпеки
_____ Андрій ЄФІМЕНКО
2 лютого 2026 р.

ЗАВДАННЯ
на випускну кваліфікаційну роботу бакалавра

Студент: Арсентьєв Іван Олександрович

Тема роботи: «Проєкт системи оцінювання вразливостей APK-файлів»

затверджена Наказом університету від 2 лютого 2026 р р. № 58/с

Термін здачі студентом закінченої роботи 26 червня 2026 р.

Вихідні дані роботи:

Предметов дослідження є методи, техніки та інструменти аналізу APK-файлів для автоматизованого виявлення вразливостей та оцінювання ризиків безпеки.

Об'єктом дослідження є процеси аналізу захищеності мобільних додатків на платформі Android, зокрема виявлення уразливостей у коді, конфігураційних файлах та архітектурі APK-файлів.

Обладнання та програмне забезпечення:

1. Windows Subsystem on Linux 2.6.3.0
2. Ubuntu Linux
3. Python 3.14.4
4. OpenJDK 17.0.19

АНОТАЦІЯ

Випускна кваліфікаційна робота бакалавра складається з пояснювальної записки, що містить вступ, три розділи, висновки, список використаної літератури та додатки. Текстова частина викладена на 63 сторінках друкованого тексту. Пояснювальна записка має 23 сторінки додатків. Список використаних джерел містить 29 найменувань і займає 3 сторінки. В роботі наведено 42 рисунка та 2 таблиці. Загальний обсяг роботи – 98 сторінок.

Метою кваліфікаційної роботи є дослідження, розробка та практичне застосування комплексної методики автоматизованого оцінювання вразливостей APK-файлів шляхом швидкого парсингу метаданих, статичного аналізу декомпільованого коду, семантичного AST-сканування та taint-трасування потоків даних.

В рамках роботи було розглянуто теоретичні аспекти безпеки архітектури ОС Android та структуру додатків. Окрему увагу приділено механізмам захисту коду, технікам обфускації, конфігураційним ризикам та архітектурним уразливостям мобільних додатків.

Виконано практичне оцінювання захищеності APK-файлу, включно з експрес-аналізом маніфесту на уразливість Janus (CVE-2017-13156), паралельною декомпіляцією через JADX та apktool, сигнатурним пошуком за допомогою regex-правил, семантичним аналізом із використанням semgrep-правила.

КЛЮЧОВІ СЛОВА: APK-ФАЙЛ, ДЕКОМПІЛЯЦІЯ, СТАТИЧНИЙ АНАЛІЗ, SEMGREP, TAINT-АНАЛІЗ, ВРАЗЛИВІСТЬ.

Консультанти з випускної кваліфікаційної роботи із зазначенням розділів, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Єфіменко А.А.	02.02.2026	02.02.2026
2	Єфіменко А.А.	02.02.2026	02.02.2026
3	Єфіменко А.А.	02.02.2026	02.02.2026

Студент _____ Іван АРСЕНТЬЄВ

Керівник _____ Артем ЛАЙЧУК

Календарний план

№ з/п	Назви етапів випускної кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка завдання	08.05.2026	Виконано
2	Розробка плану дослідження	12.05.2026	Виконано
3	Аналіз теоретичних аспектів безпеки архітектури APK-файлів	16.05.2026 – 20.05.2026	Виконано
4	Розробка методики практичного оцінювання вразливостей APK-файлів	21.05.2026 – 24.05.2026	Виконано
5	Декомпіляція та наліз коду аналіз маніфесту та дозволів вихідного коду	25.05.2026 – 02.06.2026	Виконано
6	Автоматизований сигнатурний та семантичний аналіз декомпільованого коду	03.06.2026 – 05.06.2026	Виконано
7	Аналіз потоків даних та трасування ланцюжків викликів	06.06.2026 – 10.06.2026	Виконано
8	Перевірка результатів аналізу вразливого	14.06.2026 – 16.06.2026	Виконано
9	Написання пояснювальної записки	17.06.2026 – 21.06.2026	Виконано
10	Захист	26.06.2026	

Студент _____ Іван АРСЕНТЬЄВ

Керівник _____ Артем ЛАЙЧУК

ANNOTATION

The bachelor’s thesis consists of an explanatory note containing an introduction, three chapters, conclusions, a list of references, and appendices. The main text comprises 63 pages of printed text. The explanatory note includes 23 pages of appendices. The list of references contains 29 entries and spans 3 pages. The thesis includes 42 figures and 3 tables. The total length of the thesis is 98 pages.

The purpose of this thesis is to research, develop, and practically apply a comprehensive methodology for the automated assessment of vulnerabilities in APK files through rapid metadata parsing, static analysis of decompiled code, semantic AST scanning, and taint tracing of data flows.

The thesis examines the theoretical aspects of the Android application architecture’s security and the structure of applications. Particular attention is paid to code protection mechanisms, obfuscation techniques, configuration risks, and architectural vulnerabilities in mobile applications.

A practical assessment of the security of an APK file was performed, including a quick analysis of the manifest for the Janus vulnerability (CVE-2017-13156), parallel decompilation using JADX and apktool, signature-based search using regex rules, and semantic analysis using the semgrep rule.

KEYWORDS: APK FILE, DECOMPILATION, STATIC ANALYSIS, SEMGREP, TAINT ANALYSIS, VULNERABILITY.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....7
ВСТУП.....9
РОЗДІЛ 1. ТЕОРЕТИЧНІ АСПЕКТИ БЕЗПЕКИ АРХІТЕКТУРИ APK-ФАЙЛІВ ТА МЕТОДОЛОГІЯ ЇХ АНАЛІЗУ.....11
1.1. Архітектурні принципи побудови та забезпечення безпеки операційної системи Android і структура мобільних додатків.....11
1.2. Класифікація конфігураційних ризиків та архітектурних уразливостей у мобільних додатках.....19
1.3. Світові стандарти та методології аналізу безпеки мобільного ПЗ.....24
1.4. Технологічний стек та інструментарій автоматизованого сканування.....31
РОЗДІЛ 2. МЕТОДИ СТАТИЧНОГО ТА СЕМАНТИЧНОГО ОЦІНЮВАННЯ ЗАХИЩЕНОСТІ APK-ФАЙЛІВ.....37
2.1. Концептуальний план аналізу бінарних артефактів та архітектура триетапного конвеєра.....37
2.2. Методи експрес-парсингу метаданих та сигнатурного аналізу вихідного коду.....42
2.3. Семантичний AST-аналіз та моделювання небезпечних потоків даних.....43
2.4. Поглиблений семантичний AST-аналіз та механізми інструментального трасування ланцюжків викликів.....46
РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ АВТОМАТИЗОВАНОГО КОНВЕЄРА ТА ВЕРИФІКАЦІЯ РЕЗУЛЬТАТІВ ОЦІНЮВАННЯ APK-ФАЙЛІВ.....51
3.1 Постановка завдання та вибір вразливого додатку.....51
3.2 Створення скелету пайплайна та реалізація експрес-парсингу метаданих.....52
3.3 Паралельна декомпіляція та аналіз потоків даних.....57
3.4 Сигнатурний та семантичний аналіз.....60
3.5 Перевірка результатів аналізу.....65
ВИСНОВКИ.....69
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....71
ДОДАТКИ.....74

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AOT – Ahead-of-Time

API – Application Programming Interface

APK – Android Package Kit

ART – Android Runtime

AST – Abstract Syntax Tree

AXML – Android Binary XML

CI/CD – Continuous Integration / Continuous Deployment

DES – Data Encryption Standard

DSS – Data Security Standard

EMV – Europay, MasterCard, and Visa

FFI – Foreign Function Interface

IPC – Inter-Process Communication

ISO – International Organization for Standardization

JDWP – Java Debug Wire Protocol

JNI – Java Native Interface

MASTG – Mobile Application Security Testing Guide

MASVS – Mobile Application Security Verification Standard

MD5 – Message Digest 5

MDM – Mobile Device Management

MITM – Man-in-the-Middle

NIST – National Institute of Standards and Technology

OWASP – Open Web Application Security Project

PCI – Payment Card Industry

SAPP – Static Analysis Post Processor

SAST – Static Application Security Testing

SDK – Software Development Kit

SELinux – Security-Enhanced Linux

SMS – Short Message Service

SSL – Secure Sockets Layer

VM – Virtual Machine

XML – Extensible Markup Language

ОС – операційна система

ПЗ – програмне забезпечення

ВСТУП

Актуальність теми. З кожним роком кількість користувачів мобільних пристроїв на базі ОС Android невпинно зростає, що робить мобільні додатки головною мішенню для кібератак. Через поспіх під час розробки програмісти часто не приділяють належної уваги безпеці, залишаючи в APK-файлах критичні конфігураційні помилки, жорстко закодовані API-ключі та архітектурні вразливості. Швидке виявлення таких недоліків до моменту публікації додатка є критично важливим. У межах цієї роботи вирішується проблема автоматизації процесу пошуку вразливостей шляхом створення конвеєра глибокого аналізу, що дозволяє суттєво знизити ризики витоку персональних даних користувачів та компрометації мобільних платформ.

Метою кваліфікаційної роботи є дослідження, розробка та практичне впровадження комплексної автоматизованої методики оцінювання вразливостей APK-файлів для підвищення рівня захищеності мобільних додатків на платформі Android.

Визначена мета обумовлює наступні завдання:

1. Проаналізувати теоретичні аспекти безпеки архітектури ОС Android та вивчити структуру й потенційні вразливості APK-файлів.

2. Дослідити сучасні інструменти реверс-інжинірингу, сигнатурного пошуку та семантичного AST-аналізу коду.

3. Розробити тристадійну методику (Stage 0-2) практичного оцінювання захищеності Android-додатків.

4. Реалізувати процеси автоматизованої декомпіляції, сигнатурного сканування та семантичного аналізу.

5. Налагодити механізми taint-аналізу для відстеження небезпечних потоків даних

6. Провести практичну перевірку результатів аналізу на базі вразливого APK-файлу та зафіксувати результати дослідження.

Об’єктом дослідження є процеси криміналістичного та безпекового аналізу захищеності мобільних додатків для операційної системи Android на етапі дослідження їхніх бінарних зразків.

Предметом дослідження є методи, техніки та програмні інструменти декомпіляції, сигнатурного пошуку, семантичного AST-сканування та taint-трасування для оцінювання вразливостей архітектури й коду APK-файлів.

У процесі виконання кваліфікаційної роботи було розроблено проект автоматизованого конвеєра аналізу безпеки, який здійснює швидкий парсинг метаданих маніфесту, багатопотокову декомпіляцію та багаторівневе сканування вихідного коду для виявлення критичних уразливостей у мобільних додатках Android.

РОЗДІЛ 1. ТЕОРЕТИЧНІ АСПЕКТИ БЕЗПЕКИ АРХІТЕКТУРИ АРК-ФАЙЛІВ ТА МЕТОДОЛОГІЯ ЇХ АНАЛІЗУ

1.1. Архітектурні принципи побудови та забезпечення безпеки операційної системи Android і структура мобільних додатків

Архітектура безпеки операційної системи Android формувалася еволюційним шляхом, запозичуючи надійні низькорівневі механізми захисту з ядра Linux та адаптуючи їх до специфіки мобільного середовища виконання. Головною парадигмою, яка визначає ізоляцію програмних компонентів у цій операційній системі, є концепція безпекового обмеження, відома у світовій практиці як Application Sandboxing [1]. Дана концепція передбачає, що кожне програмне забезпечення розглядається системою як окремий, потенційно ненадійний суб'єкт, якому під час інсталяції присвоюється унікальний ідентифікатор користувача ядра [2]. Такий підхід дозволяє повністю розмежувати права доступу на рівні процесів: кожен додаток функціонує у своєму власному ізольованому віртуальному просторі, не маючи жодної апаратної чи програмної можливості здійснювати пряме зчитування, модифікацію або знищення даних інших програмних продуктів, встановлених у системі, якщо на це немає чітко вираженої згоди користувача або розробника.

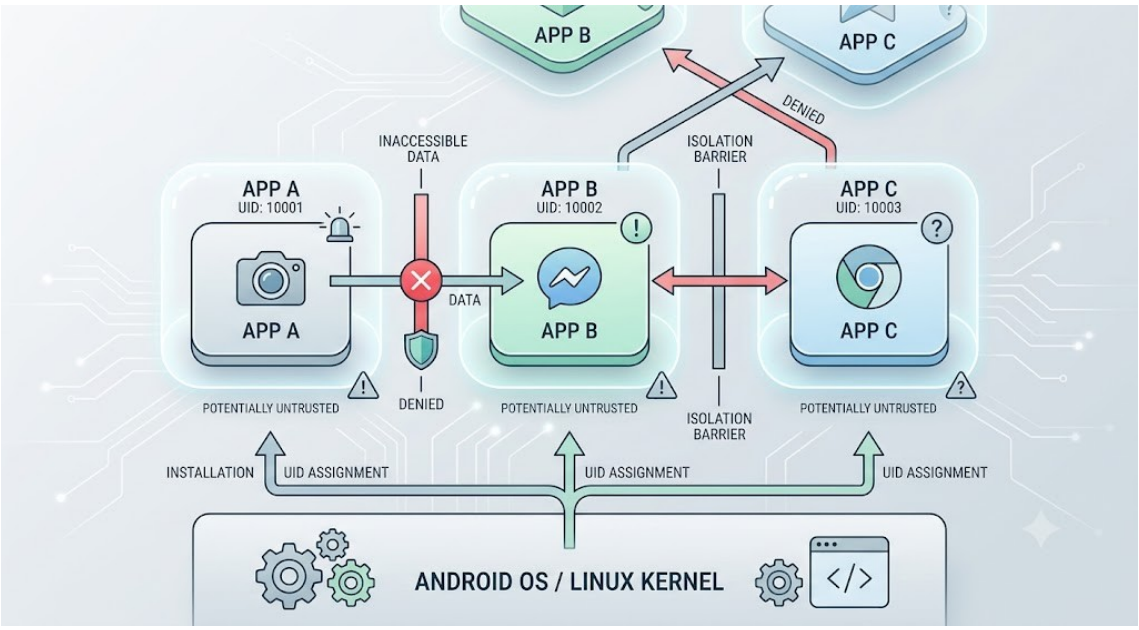


Рисунок 1.1.1 – Ізоляція програмних компонентів в Android

Додатковим і надзвичайно потужним бар'єром захисту, який функціонує на найнижчих рівнях операційної системи, є підсистема примусового контролю доступу SELinux. Вона реалізує жорстку політику безпеки, засновану на принципі мінімальних привілеїв, що дозволяє суттєво обмежити потенційні дії навіть тих процесів, які теоретично змогли отримати найвищі адміністративні права. Це мінімізує ризики успішного проведення горизонтальних атак або несанкціонованого підвищення привілеїв у системі [3]. Проте, взаємодія між різними додатками, а також їхній доступ до критичних системних ресурсів регулюється за допомогою гнучкої системи дозволів. Ці дозволи проходять ретельну класифікацію за рівнями загрози – від звичайних, що надаються автоматично під час інсталяції, до небезпечних, які вимагають динамічного підтвердження з боку користувача безпосередньо в момент запиту під час виконання програми. Окреме місце посідають дозволи рівня підпису, які надаються операційною системою автоматично лише в тому випадку, якщо додаток, що запитує доступ, підписаний тим самим цифровим сертифікатом розробника, що й додаток, який цей дозвіл оголосив [4].

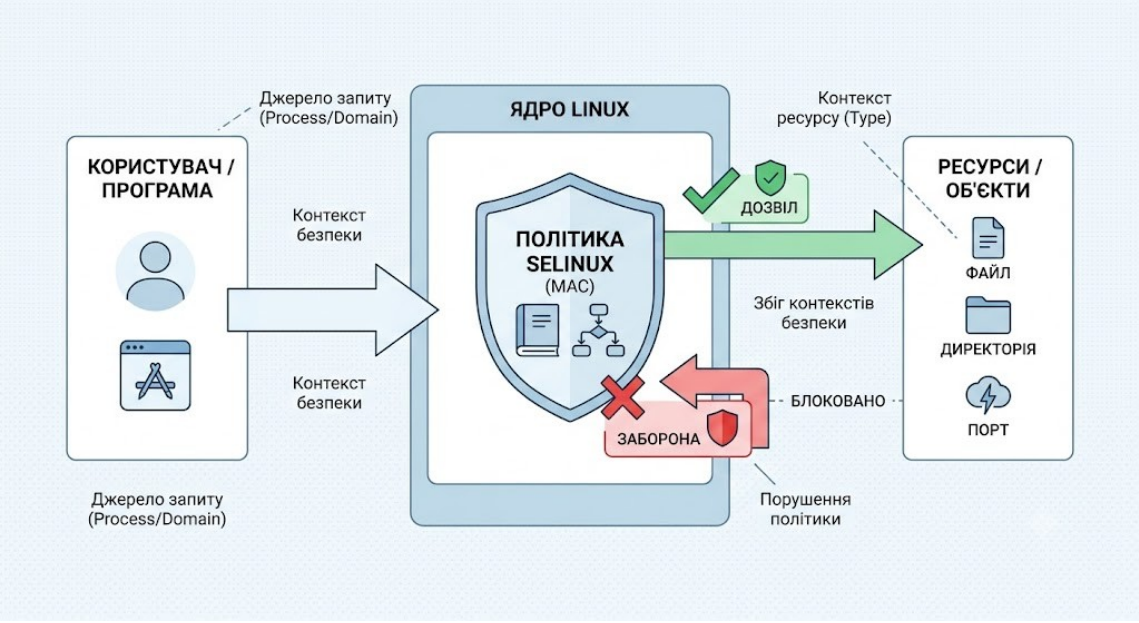


Рисунок 1.1.2 – Принцип роботи SELinux

Говорячи про легітимність та перевірку цілісності програмного забезпечення, неможливо оминути процес цифрового підписання APK-файлів. Протягом

тривалого часу компанія Google послідовно розробляла та впроваджувала нові схеми підписів для запобігання несанкціонованій модифікації коду:

1. Схема v1, заснована на перевірці окремих файлів всередині архіву, виявилася вразливою до підміни структури самого пакета;

2. Схема v2 запропонувала захист усього бінарного файлу за допомогою спеціального блоку підпису , що значно прискорило процес верифікації;

3. Схема v3 розширила ці можливості, додавши механізм ротації сертифікатів , що дозволяє розробникам безпечно оновлювати ключі, зберігаючи повну історію попередніх підписів [5].

З технічної та структурованої точок зору, файл із розширенням .apk являє собою спеціалізовану модифікацію стандартного ZIP-архіву, який має суворо визначену внутрішню анатомію. Центральне місце у цій структурі посідає бінарний файл конфігурації AndroidManifest.xml, який виступає в ролі головного архітектурного паспорта програми. На відміну від текстового формату XML, який використовується під час розробки, всередині готового інсталяційного пакета цей документ зберігається у скомпільованому бінарному вигляді задля оптимізації його зчитування операційною системою. Він містить у собі абсолютно всю ключову декларативну інформацію про додаток, включаючи унікальну назву пакета, обмеження щодо мінімального та цільового рівнів Android SDK, повний перелік необхідних для функціонування дозволів, а також опис та конфігурацію основних компонентів системи, таких як Activities, Services, Broadcast Receivers та Content Providers, разом із налаштуваннями їхньої експозиції для міжпроцесної взаємодії.

Сама ж логіка програми та вихідний код компілюються у файли формату .dex, які оптимізовані для виконання віртуальною машиною Android Runtime або її застарілим аналогом Dalvik. Тісно пов'язаною із байт-кодом є таблиця скомпільованих ресурсів resources.arsc, яка містить бінарну структуру ідентифікаторів, що пов'язує змінні з файлів .dex із конкретними статичними елементами, такими як рядкові константи різними мовами, колірні палітри або шляхи до графічних об'єктів [6]. Дослідження цієї таблиці є критично важливим

етапом аудіту безпеки для виявлення зашитих у коді API-ключів, токенів доступу або адрес серверів керування.

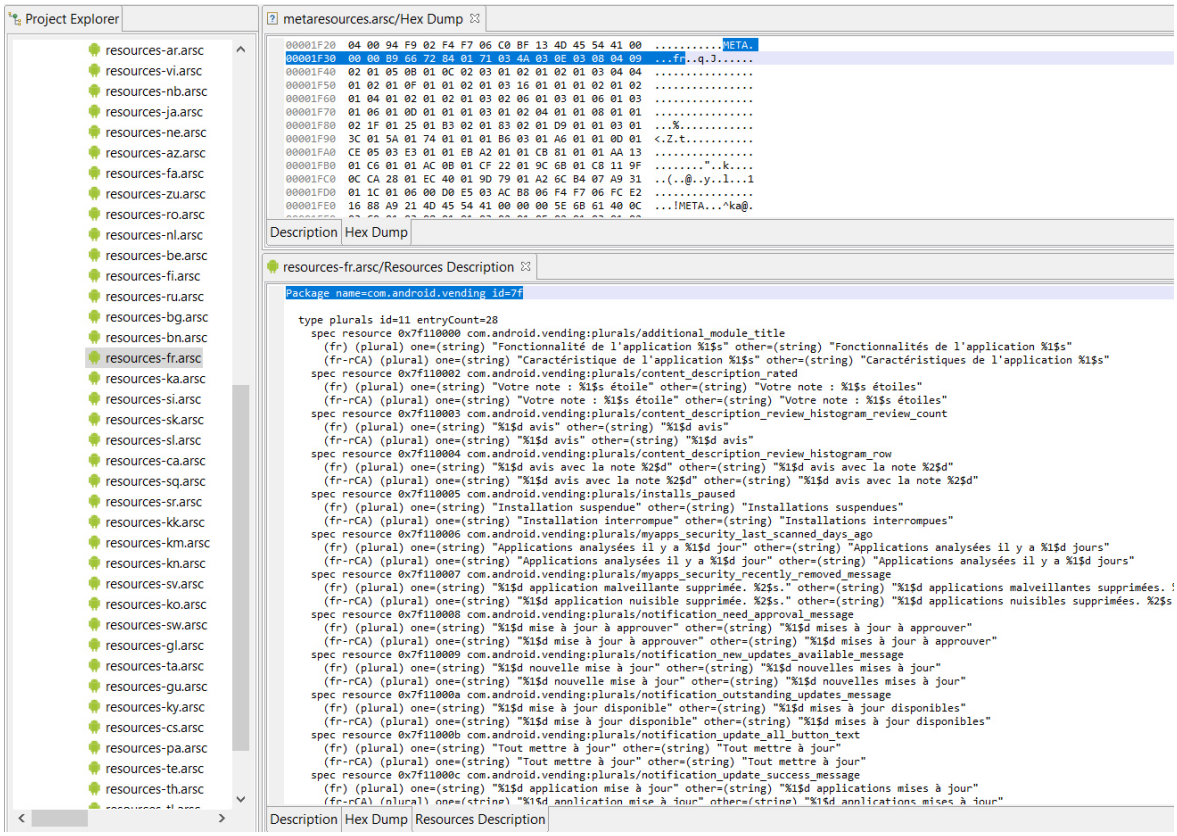


Рисунок 1.1.3 – Приклад файлу resources.arsc

Графічне, інтерфейсне та медіа-наповнення додатка розподіляється між двома основними директоріями, якими є res/ та assets/. Каталог res/ містить структуровані ресурси, які були оброблені компілятором і отримали унікальні ідентифікатори в таблиці ресурсів, тоді як папка assets/ призначена для збереження "сирих" файлів, кастомних конфігурацій або локальних баз даних, доступи до яких здійснюються за їхніми текстовими іменами за допомогою менеджера ресурсів. Зловмисники часто використовують директорію assets/ для прихованого зберігання зашифрованих шкідливих модулів або динамічних завантажувачів. Для захисту від подібних модифікацій та забезпечення автентичності додатка в архітектурі APK передбачено каталог криптографічного контролю META-INF/. Ця папка містить маніфести хеш-сум усіх файлів архіву, файли підпису та цифрові сертифікати розробника, що дозволяє операційній системі верифікувати цілісність пакета і переконатися, що вміст інсталятора не був змінений сторонніми особами після підписання автором [7].

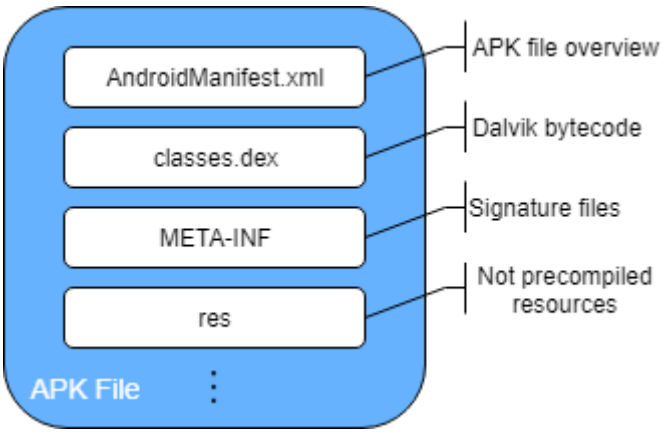


Рисунок 1.1.4 – Структура APK

Окремий рівень складності для аналізу безпеки становить папка нативних бінарних бібліотек lib/, де розміщуються динамічні бібліотеки з розширенням .so, розроблені за допомогою Android NDK на низькорівневих мовах програмування C або C++. Структура цієї папки традиційно розділена на піддиректорії, що відповідають конкретним архітектурам процесорів, таким як armeabi-v7a, arm64-v8a, x86 та x86_64. Наявність нативного коду вимагає застосування спеціалізованих інструментів для дизасемблювання, оскільки такий код виконується в обхід віртуальної машини ART безпосередньо на процесорі пристрою, що відкриває можливості для реалізації низькорівневих атак типу переповнення буфера або ручного приховування шкідливої логіки від стандартних автоматизованих засобів статичного аналізу додатків [8].

Головним сполучним елементом, який забезпечує технологічну взаємодію між високорівневим кодом Java/Kotlin та цими низькорівневими бінарними структурами, є архітектурний інтерфейс JNI. . Цей міст дозволяє коду, що виконується всередині віртуальної машини Android Runtime, прозоро викликати функції, скомпільовані в .so бібліотеках, і навпаки – дозволяє нативному коду маніпулювати об'єктами Android-додатка в обхід стандартних безпекових обмежень високого рівня. З точки зору аналізу безпеки, саме в місцях реалізації інфраструктури JNI виникає величезна кількість критичних дефектів, оскільки неправильна валідація даних під час переходу через цей інтерфейс призводить до руйнування меж ізоляції пам'яті. Зловмисники та розробники шкідливого програмного забезпечення часто використовують JNI як інструмент обфускації,

виносячи туди чутливі операції – такі як динамічне завантаження сторонніх файлів через метод RegisterNatives, розшифрування прихованих конфігурацій чи перевірку середовища на наявність рут-прав. Дослідження таких переходів суттєво ускладнює реверс-інжиніринг, оскільки експерту з кібербезпеки доводиться не просто аналізувати логіку бінарного файлу в декомпіляторах на кшталт Ghidra або IDA Pro, а й вручну відновлювати складні таблиці відповідностей та сигнатур методів, які зв'язують оголошені у маніфесті чи коді додатка ключові слова native із їхньою реальною низькорівневою реалізацією у машинних інструкціях цільового процесора [9].

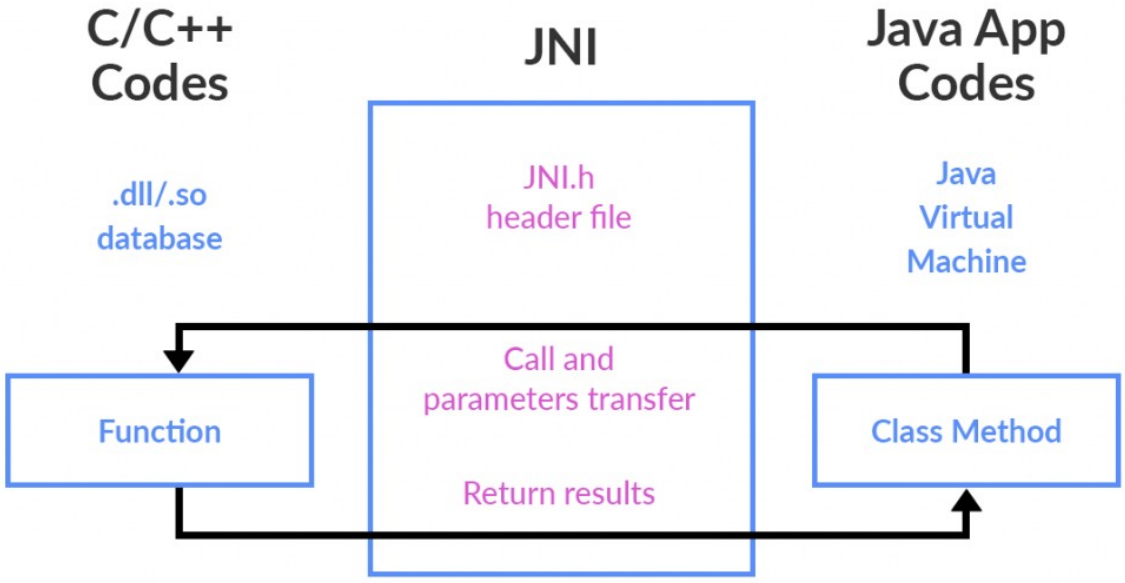


Рисунок 1.1.5 – Принцип взаємодії нативного кода з Java/Kotlin через JNI

У сучасній індустрії мобільної розробки під операційну систему Android особливе місце посідає Flutter – відкритий кросплатформовий фреймворк від компанії Google, призначений для створення високопродуктивних додатків із кастомізованим графічним інтерфейсом. На відміну від класичного підходу, де інтерфейс будується на базі системних компонентів ОС, Flutter функціонує як потужний графічний рушій, що самостійно відмальовує кожен піксель на екрані за допомогою бібліотеки Impeller або Skia. Ця технологія використовується для значного прискорення процесу розробки, оскільки дозволяє писати єдину кодову базу для кількох платформ одночасно, забезпечуючи при цьому швидкість роботи та візуальний відгук на рівні нативних застосунків. Проте використання Flutter та

його базової мови програмування Dart вносить корінні зміни у звичну парадигму забезпечення та аналізу безпеки мобільних додатків.

Головна технологічна відмінність релізних збірок Flutter в Android-середовищі полягає у повній відмові від інтерпретації коду або використання проміжного байт-коду, характерного для класичної розробки на Java чи Kotlin. Замість цього застосовується режим компіляції AOT, який повністю трансформує вихідний код Dart у монолітний масив нативних машинних інструкцій під конкретні цільові архітектури процесорів. З точки зору інженерії безпеки, такий підхід забезпечує потужний рівень природної обфускації, оскільки фінальний компілят не містить людиночитаних метаданих, таблиць символів, назв класів чи сигнатур методів. Це робить стандартні декомпілятори та утиліти автоматизованого реверс-інжинірингу мобільного софту абсолютно неефективними, створюючи серйозний бар'єр для зловмисників, які намагаються дослідити інтелектуальну власність або логіку роботи додатка.

Проте, високий уровень захисту від швидкого декомпілювання породжує специфічні виклики для проведення внутрішнього аудиту та статичного аналізу безпеки. Оскільки весь прикладний код разом із інтегрованою віртуальною машиною Dart VM постачається у вигляді закритих нативних бібліотек, традиційні сканери уразливостей мобільних додатків опиняються у «сліпій зоні». Вони не здатні проаналізувати наскрізні потоки даних або виявити декларативні дефекти всередині скомпільованого Dart-коду. Для верифікації відсутності вразливостей або прихованих шкідливих закладок аналітикам доводиться застосовувати складні інструменти низькорівневого аналізу, такі як дизасемблери та інтерактивні декомпілятори класу Ghidra або IDA Pro, використовуючи спеціалізовані плагіни для парсингу внутрішньої структури Dart Snapshot, ручного відновлення зміщень у пам'яті та реконструкції об'єктної моделі середовища виконання.

Окремим критичним аспектом безпеки в екосистемі Flutter є механізм взаємодії із системними функціями пристрою, який реалізується через Platform Channels та інтерфейс Dart FFI. Оскільки Flutter-додаток працює у власному ізольованому середовищі, будь-який доступ до сенсорів, файлової системи,

захищених сховищ або нативного коду ОС вимагає передачі повідомлень через асинхронні канали. Безпека цієї архітектури повністю покладається на коректність фільтрації та валідації даних у точках перетину кордонів між середовищем Dart та нативним кодом платформи. Помилки проєктування, недостатня санітизація вхідних параметрів або неконтрольована експозиція цих каналів створюють ризики впровадження стороннього шкідливого коду, витоку конфіденційної інформації та порушення ізоляції пам'яті, що вимагає ретельного проектування безпеки на етапі створення архітектури додатків.



Рисунок 1.1.6 – Архітектура Flutter додатку

1.2. Класифікація конфігураційних ризиків та архітектурних уразливостей у мобільних додатках

Аналіз захищеності мобільних екосистем та компілятів прикладного програмного забезпечення вимагає фундаментального, системного підходу до типізації та класифікації векторів деструктивного впливу. Сучасна парадигма мобільної безпеки оперує складними багатокомпонентними структурами, де межа між системними помилками адміністрування, поверхневими конфігураційними огріхами та концептуальними, фундаментальними дефектами проєктування є надзвичайно дифузною та розмитою. Це зумовлено специфікою самих мобільних операційних систем, які поєднують у собі складні механізми ізоляції процесів на низькому рівні із високорівневими прикладними інтерфейсами, що створює широке поле для потенційного виникнення вразливостей на стику технологічних рівнів. Своєчасна ідентифікація та правильна класифікація таких ризиків є критично важливим етапом превентивного захисту, оскільки характер усунення дефекту напряму залежить від його природи: якщо конфігураційні помилки часто можна виправити зміною параметрів у файлах маніфесту або налаштуваннях середовища збирання, то архітектурні вразливості вимагають глибокого рефакторингу вихідного коду та перегляду логіки взаємодії компонентів.

Для подолання хаотичності у визначенні загроз та формування єдиної термінологічної бази в індустрії кібербезпеки, більшість ключових загроз та дефектів були ретельно каталогізовані, структуровані та стандартизовані у глобальних аналітичних фреймворках. Базовим та найавторитетнішим орієнтиром для світової спільноти розробників, аудиторів та інженерів із захисту інформації слугує міжнародний стандарт OWASP Mobile Top 10, який регулярно оновлюється на основі статистичних даних про реальні інциденти та консалтингову практику провідних лабораторій світу. Цей фреймворк не просто перелічує критичні вразливості, а групує їх за векторами виникнення, оцінюючи технічні наслідки для бізнесу та кінцевих користувачів, починаючи від неналежного використання платформи та закінчуючи недостатньою криптографічною стійкістю чи незахищеним зберіганням чутливих даних. Використання такої стандартизованої

класифікації дозволяє дослідникам вибудовувати методологічно правильні стратегії статичного та динамічного аналізу, забезпечуючи системне покриття всіх можливих вразливих зон мобільного додатка під час перевірки його захищеності.

Проте, як свідчить глибокий ретроспективний аналіз практичної діяльності провідних консалтингових компаній та лабораторій дослідження кіберзагроз, детермінація виникнення переважної більшості вразливостей лежить не у площині застосування зловмисниками інноваційних, раніше невідомих технологій класу Zero-Day. Першопричиною є системний антропогенний фактор: комерційний тиск ринку, прагнення мінімізувати показник Time-to-Market, а також стійке ігнорування або редукція базових принципів безпечного життєвого циклу розробки програмного забезпечення на етапах архітектурного моделювання та безпосереднього кодування [10].

Конфігураційні ризики та архітектурні дефекти мобільних додатків доцільно класифікувати за двома макровекторами: ризики рівня середовища виконання та збірк та дефекти логіко-структурної побудови системного коду. Конфігураційні помилки виникають на етапі декларативного опису параметрів функціонування додатка у метаданих та інструкціях для операційної системи. Вони трансформують потенційно безпечний вихідний код у вразливий бінарний артефакт.

Один з типів конфігураційних ризиків це активація налагоджувальних дескрипторів у комерційних релізах є критичним порушенням режиму ізоляції процесів. Цей стан деструктуризує захисні бар'єри операційної системи, оскільки офіційно санкціонує підключення зовнішніх інструментів динамічного аналізу через підсистеми транспортування на зразок JDWP, дозволяє сторонньому суб'єкту з низькими привілеями здійснювати мапування адресного простору процесу в режимі runtime та створює умови для ін'єкцій довільного коду в контексті легітимного ідентифікатора користувача, нівелюючи концепцію "пісочниці" [11].

До конфігураційних ризиків також відносять деградацію політик мережевого транспорту та криптографічного захисту інваріантів. Некоректна архітектурна конфігурація або повне ігнорування декларативних маніфестів безпеки мережі призводить до компрометації канального та сеансового рівнів передачі даних.

Систематичне внесення виключень, що дозволяють використання відкритих, нетекстових та нешифрованих протоколів, а також примусове відключення процедур валідації ланцюжків сертифікатів, створює стійкі передумови для реалізації атак класу MITM. За таких умов зловмисник отримує можливість не лише пасивного перехоплення конфіденційного трафіку, а й активної сурогатної модифікації об'єктів даних у процесі їх транзиту [12].

Останній тип конфігураційних ризиків це ескалація надлишкового контексту привілеїв. Запит надлишкового пулу дозволів, які виходять за межі мінімально необхідного функціонального базису додатка, суттєво розширює потенційну поверхню атаки. У разі успішної експлуатації будь-якої іншої вразливості всередині компонента, зловмисник автоматично успадковує всі легітимно отримані раніше права доступу до апаратних та програмних ресурсів пристрою, що дозволяє використовувати скомпрометований додаток як проксі-вузол для подальшої компрометації всієї ОС [13].

На відміну від конфігураційних ризиків архітектурні дефекти закладаються на етапі проектування логіки взаємодії компонентів, вибору схем підписання бінарних файлів та стратегій керування секретами. Вони є фундаментальними вадами логіки додатка.

Одним з архітектурних дефектів у Android-додатках є неконтрольована експозиція інтерфейсів міжпроцесної взаємодії. Сучасні мобільні операційні системи оперують модульною структурою компонентів. Оголошення цих сутностей як експортованих без накладання жорстких обмежувальних декларацій або кастомних дозволів створює критичні архітектурні прогалини. Будь-яке стороннє програмне забезпечення, що функціонує в аналогічному користувацькому просторі, може генерувати неавторизовані інтенційні запити, перехоплювати широкомовні повідомлення, ініціювати приховані фонові процеси або здійснювати пряме читання/запис через незахищені контент-провайдери [14]. Це призводить до витоку контексту безпеки та несанкціонованого керування бізнес-логікою.

Іншим архітектурним дефектом є неправильна валідація бінарної цілісності та контейнеризації. Прикладом глибоких системних збоїв у логіці перевірки автентичності є уразливості сімейства Janus та аналогічні їм дефекти, що виникають внаслідок використання застарілих або редукованих схем цифрового підпису бінарних файлів. Суть уразливості полягає в архітектурній особливості зчитування файлу операційною системою: додаток може одночасно інтерпретуватися і як дійсний поштовий архів APK, і як виконуваний файл формату DEX. Це дозволяє зловмиснику здійснити операцію конкатенації шкідливого байт-коду з легітимним контейнером без руйнування криптографічного підпису першої версії, забезпечуючи приховане оновлення або заміну оригінального додатка на скомпрометований аналог з повним збереженням локальних даних користувача [15].

Найлегшим у експлуатації архітектурним дефектом є статична компрометація автентифікаційних ідентифікаторів. Пряме включення конфіденційних атрибутів – таких як статичні API-ключі сторонніх сервісів, приватні ключі асиметричного шифрування, токени доступу адміністраторів або паролі доступу до локальних баз даних – у вихідний код, константні пули або ресурсні XML-файли є критичним архітектурним недоліком. З огляду на те, що компільовані мобільні додатки за своєю природою передаються на клієнтську сторону, вони є об'єктом неминучого реверс-інжинірингу. Застосування базових інструментів декомпіляції дозволяє зловмиснику в автоматизованому режимі вилучити всі статичні секрети, що призводить до повної компрометації пов'язаної серверної інфраструктури або інтегрованих хмарних API [16].

Для наочної систематизації та кореляції вищеописаних деструктивних факторів сформовано аналітичну матрицю, яка відображає зв'язок між типом помилки, її технологічним доменом та потенційними наслідками для системи.

Таблиця 1.2.1 – Таблиця типових ризиків у Android-додатку

Категорія ризику	Конкретний дефект	Технологічний домен	Вектор експлуатації / Наслідки	Рекомендований метод нівелювання
Конфігураційний	debuggable="true"	Метадані збірки	Динамічна ін'єкція коду, рантайм-інтроспекція пам'яті через JDWP.	Автоматизація збірки через лінтери (ProGuard, R8, Gradle build types).
Конфігураційний	Cleartext Traffic Allowed	Мережевий стек транспорту	Перехоплення даних, модифікація API-відповідей.	Впровадження суворих декларацій `NetworkSecurityConfig`, Certificate Pinning.
Конфігураційний	Надлишкові Permissions	Політика доступу ОС	Розширення поверхні атаки, несанкціонований доступ до сенсорів.	Дотримання принципу мінімальних привілеїв, динамічний запит прав.
Архітектурний	Незахищений IPC-експорт	Компоненти програми	Неавторизований виклик функцій сторонніми додатками, витік даних.	Декларація внутрішніх дозволів, обмеження експорту.
Архітектурний	Застаріла схема підпису	Бінарний контейнер	Підміна байт-коду зі збереженням легітимності підпису.	Обов'язкове використання схем підпису нових поколінь (v2, v3, v4).
Архітектурний	Hardcoded Secrets	Вихідний код / Ресурси	Реверс-інжиніринг, масовий витік ключів доступу до хмарних інфраструктур.	Винесення секретів у Vault/Keystore, динамічна автентифікація.

Таким чином, розбудова стійкої архітектури мобільного додатка вимагає комплексного підходу, де параметри конфігурації середовища та логіка побудови коду розглядаються як єдиний пов'язаний комплекс оборонних рубежів.

1.3. Світові стандарти та методології аналізу безпеки мобільного ПЗ

Динамічний розвиток екосистеми мобільних технологій та їх глибока інтеграція у повсякденне життя, фінансовий сектор, корпоративне управління та критичну інфраструктуру висувають безпрецедентні вимоги до забезпечення інформаційної безпеки мобільних додатків. На відміну від традиційного веб- або десктопного програмного забезпечення, мобільні додатки функціонують у специфічному, гетерогенному та часто ворожому середовищі. Вони виконуються на кінцевих пристроях користувачів, де операційна система може бути скомпрометована, піддана процедурам отримання суперкористувацьких прав або працювати під контролем сторонніх шкідливих модулів. Крім того, архітектура мобільного пакета передбачає зберігання значної частини логіки, конфігурацій та бінарних ресурсів на боці клієнта, що робить його вразливим до зворотного проектування, модифікації та підміни коду.

У зв’язку з цим виникла гостра потреба у формуванні єдиних, стандартизованих та загальновизнаних у світі методологічних підходів до оцінювання вразливостей, тестування на проникнення та забезпечення гарантій безпеки мобільного софту. Без використання уніфікованих стандартів аналіз безпеки перетворюється на хаотичний пошук окремих дефектів, який не здатний гарантувати цілісну захищеність системи. Сучасна індустрія кібербезпеки спирається на декілька фундаментальних методологій та стандартів, розроблених провідними міжнародними консорціумами та інституціями. Основними серед них є ініціативи OWASP, стандарти NIST, критерії безпеки платформ, а також адаптовані галузеві регуляції, такі як PCI DSS для фінансових додатків. Розглянемо детально структуру, еволюцію та практичне застосування цих світових стандартів у контексті аналізу безпеки мобільних додатків.

Історично найбільш впливовою організацією у сфері безпеки прикладного програмного забезпечення є Open Web Application Security Project відкритий міжнародний проект, що об'єднує експертів з усього світу з метою підвищення захищеності програмного забезпечення. Початково сфокусований на веб-додатках,

проект OWASP швидко зреагував на мобільну революцію, створивши окремий вектор досліджень – OWASP Mobile Security Project.

Головним завданням цього напрямку стало переосмислення парадигми загроз. Якщо у веб-додатках основний фокус захисту зміщений на серверний бік, то в мобільній сфері вектор атак суттєво змістився в бік клієнтського пристрою. Для систематизації найбільш критичних ризиків було створено консенсусний документ – OWASP Mobile Top 10. Цей перелік регулярно оновлюється на основі статистичних даних про реальні інциденти, результатів аудитів безпеки та експертних опитувань.

Протягом тривалого часу базовим орієнтиром виступав реєстр OWASP Mobile Top 10 , який класифікував загрози за наступними векторами:

1. M1: Improper Platform Usage – неправильне використання можливостей платформи, наприклад, некоректне налаштування дозволів Android Permissions або Intents;

2. M2: Insecure Data Storage – небезпечне зберігання даних, включаючи витік чутливої інформації через системні логи, кеш або незашифровані локальні бази даних;

3. M3: Insecure Communication – небезпечні канали зв'язку, відсутність сертифікації або вразливість до атак типу Man-in-the-Middle;

4. M4: Insecure Authentication – слабка або відсутня автентифікація користувача на боці клієнта або сервера;

5. M5: Insufficient Cryptography – недостатня або застаріла криптографія, використання слабких шифрів, наприклад, MD5 або DES;

6. M6: Insecure Authorization – проблеми авторизації та розмежування прав доступу;

7. M7: Client Code Quality – низька якість клієнтського коду, що призводить до переповнення буфера або витоку пам'яті;

8. M8: Code Tampering – вразливість до модифікації та підробки коду пакета;

9. M9: Reverse Engineering – слабкий захист від декомпіляції та аналізу логіки додатка;

10. M10: Extraneous Functionality – наявність стороннього або надлишкового функціоналу, залишеного розробниками для тестування.

Зі зміною архітектури мобільних ОС та появою нових механізмів захисту класифікація OWASP Mobile Top 10 зазнала суттєвої трансформації та укрупнення у редакції 2024 року. Сучасний перелік відображає більш комплексні архітектурні ризики, фокусуючись на вразливостях ланцюжків постачання, витоку облікових даних через хмарні інтеграції, а також на недостатній стійкості до автоматизованого збору даних.

Проте, сам по собі перелік Top 10 є лише індикатором ризиків, а не детальною інструкцією для тестування. Для вирішення завдання прикладного аналізу безпеки OWASP розробив дворівневу екосистему стандартів: OWASP MASVS та OWASP MASTG.

OWASP MASVS – це фундаментальний стандарт, який визначає чіткі архітектурні вимоги до безпеки мобільних додатків для платформ Android та iOS. Він виконує роль базової метрики, за допомогою якої розробники, архітектори та експерти з безпеки можуть кількісно та якісно оцінити рівень захищеності програмного продукту [17].

MASVS пропонує гнучку модульну структуру, розділену на чотири ключові домени безпеки, кожен з яких покриває конкретний аспект життєвого циклу додатка та обробки даних:

1. MASVS-STORAGE: Фокусується на захисті чутливих даних користувача та організації на пристрої. Стандарт вимагає, щоб жодна конфіденційна інформація не зберігалася у відкритому вигляді в пісочниці додатка, не потрапляла до системних логів, не залишалася в тимчасових файлах кешу або базах даних SQLite без належного шифрування.

2. MASVS-CRYPTO: Визначає правила використання криптографічних примітивів. Цей домен забороняє розробку власних, неперевірених алгоритмів шифрування, вимагає застосування виключно сучасних та стійких стандартів та

регламентує безпечне керування життєвим циклом ключів із використанням апаратного рівня абстракції пристрою.

3. MASVS-NETWORK: Регулює захист даних під час їх транзиту між мобільним клієнтом та серверною частиною. Вимоги цього блоку включають обов'язкове використання протоколів TLS актуальних версій, правильну конфігурацію перевірки сертифікатів та впровадження механізмів суворої прив'язки сертифікатів, що дозволяє запобігти перехопленню трафіку навіть у випадку компрометації системних центрів сертифікації.

4. MASVS-RESILIENCE: Унікальний домен мобільного стандарту, присвячений стійкості додатка до реверс-інжинірингу, динамічного аналізу, налагодження та несанкціонованої модифікації. Він містить вимоги щодо впровадження обфускації коду, механізмів виявлення Root-прав, детектування запусків в емуляторах, контролю цілісності власного двійкового коду та блокування підключення динамічних інструментів трасування типу Frida.

Якщо MASVS встановлює архітектурні цілі, то OWASP MASTG є всеохоплюючою технічною інструкцією, яка детально описує процеси, інструменти та команди для перевірки кожної з цих вимог на практиці. Це фундаментальна книга рецептів для інженерів з безпеки та пентестерів. MASTG пропонує два взаємодоповнюючі технічні підходи до оцінювання вразливостей, які в сукупності забезпечують максимальне покриття кодової бази додатка [18].



Рисунок 1.3.1 – Різниця між OWASP MASVS та OWASP MASTG

Хоча OWASP є безумовним лідером у сфері прикладного тестування, існують інші фундаментальні державні та міждержавні стандарти, які формують методологічну базу для проведення сертифікації мобільного програмного

забезпечення, особливо в урядових установах, військовому секторі та великих транснаціональних корпораціях. Національний інститут стандартів і технологій США розробив спеціалізовану настанову NIST SP 800-163 Rev. 1. Цей документ фокусується на процесі комплексного дослідження мобільних додатків перед їх розгортанням у корпоративному або урядовому середовищі [19]. Методологія NIST базується на концепції побудови конвеєра перевірки додатків і виділяє наступні ключові типи аналізу:

1. Аналіз відповідності політикам безпеки: Оцінювання того, чи відповідає додаток внутрішнім регламентам організації.

2. Поведінковий аналіз: Специфічний підхід, орієнтований на виявлення прихованих або шкідливих функцій програми, які можуть не класифікуватися як класичні вразливості коду. Поведінковий аналіз досліджує, чи не намагається калькулятор отримати доступ до контактної книги, чи не відправляє SMS у фоновому режимі та чи не здійснює несанкціонований збір телеметрії. NIST також наголошує на необхідності використання комбінованих інструментів і детально описує критерії прийняття ризиків для організацій, що дозволяє структурувати результати автоматизованого сканування у вигляді чітких метрик ризику.

Для високорівневої сертифікації програмно-апаратних комплексів та критично важливого софту застосовується міжнародний стандарт Common Criteria. У контексті мобільних технологій розробляються спеціалізовані профілі захисту, наприклад, Protection Profile for Mobile Applications. Цей стандарт висуває жорсткі математично верифіковані вимоги до ізоляції процесів, криптографічного захисту ядра додатка та механізмів автентифікації, і використовується переважно для сертифікації мобільних операційних систем, MDM-систем та криптографічних токенів [20].

Окреме місце в методології аналізу посідають стандарти, розроблені консорціумами фінансової індустрії. Мобільний банкінг, цифрові гаманці та додатки, що перетворюють смартфон на платіжний термінал, є мішенню номер один для фінансово вмотивованих кіберзлочинців. У сфері обробки даних платіжних карток діє суворий стандарт PCI DSS. Оскільки мобільні пристрої

вважаються за умовчанням незахищеним середовищем, Рада стандартів безпеки індустрії платіжних карток випустила специфічні вимоги для мобільних додатків, що здійснюють приймання платежів [21]. Згідно з цими настановами, методологія аналізу фінансового додатка повинна обов'язково включати:

1. Перевірку неможливості перехоплення та збереження даних магнітної смуги або EMV-чіпа в пам'яті пристрою;

2. Наявність надійного наскрізного шифрування від моменту зчитування або введення даних картки на смартфоні до моменту їх отримання платіжним шлюзом;

3. Обов'язкове впровадження технологій захисту від налагодження та аналізу пам'яті, що корелює з рівнем MASVS-R від OWASP.

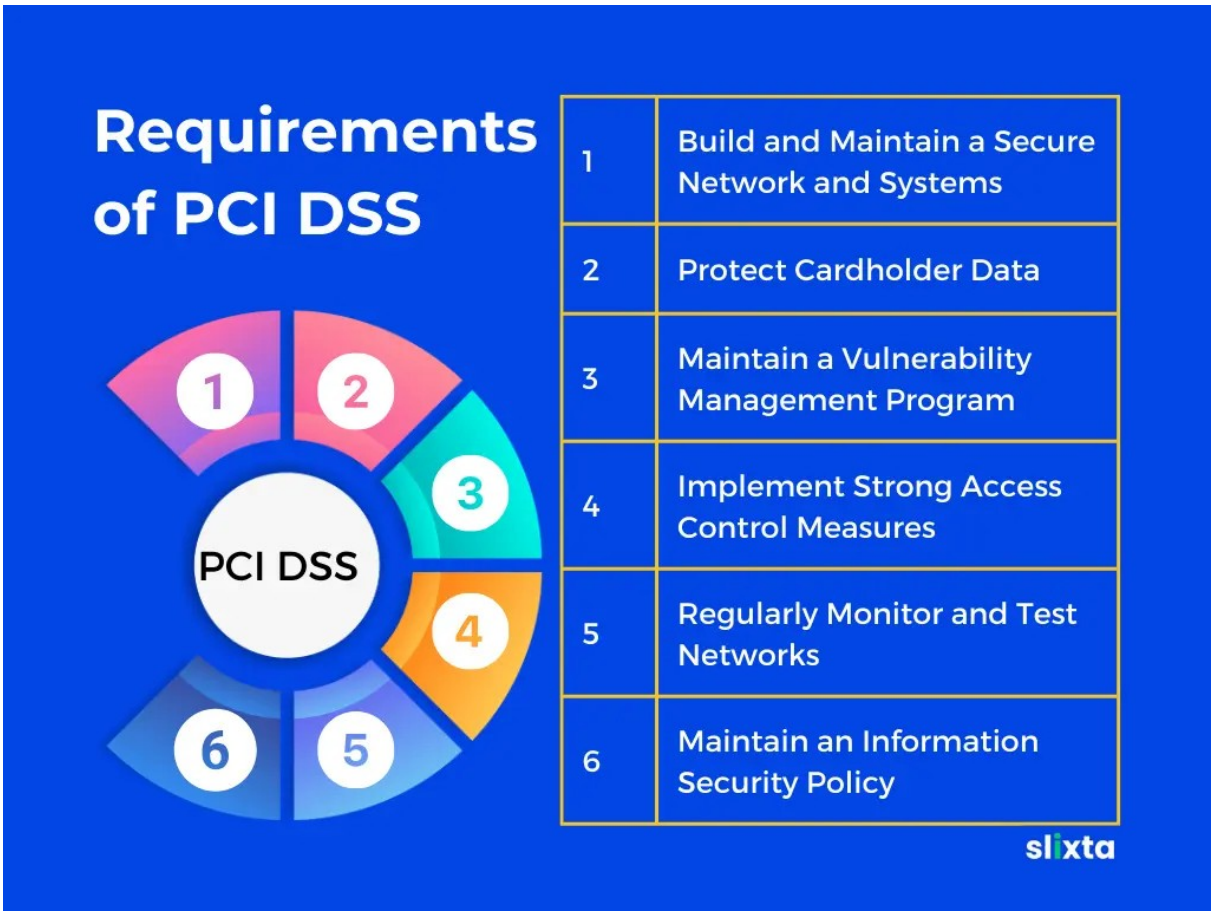


Рисунок 1.3.2 – Вимоги стандарту PCI DSS

Для вибору оптимальної стратегії оцінювання вразливостей в рамках практичного проекту необхідно провести порівняльний аналіз розглянутих методологій. Кожна з них має свою специфіку, унікальні переваги та обмеження.

Таблиця 1.3.1 – Порівняльний аналіз світових методологій

Критерій порівняння	OWASP MASVS / MASTG	NIST SP 800-163 Rev. 1	Common Criteria (ISO 15408)	PCI Mobile Guidelines
Основна ціль	Всебічний технічний аналіз безпеки та пентестинг	Впровадження додатків у корпоративне середовище	Високорівнева державна сертифікація	Захист фінансових та платіжних даних
Цільова аудиторія	Розробники, інженери з безпеки, пентестери	Системні адміністратори, офіцери безпеки	Державні органи, розробники критичних систем	Аудитори, розробники фінтех-софту
Рівень деталізації	Дуже високий	Середній	Високий	Високий (специфічний для платіжних операцій)
Фокус на платформах	Специфічний для Android та iOS	Загально-платформний	Абстрактний	Загально-платформний
Практичність автоматизації	Легко інтегрується у CI/CD процеси та сканери	Орієнтований на побудову внутрішніх процесів організації	Складно автоматизувати, потребує паперового аудиту	Орієнтований на чек-листи та регулярні аудити

Аналізуючи дані таблиці, можна зробити остаточний висновок, що для розробки та функціонування автоматизованих засобів сканування APK-файлів у межах даного проекту не підійшов жоден із чинних світових стандартів у своєму чистому вигляді. Кожна з розглянутих методологій має фундаментальні архітектурні обмеження: вони є або занадто громіздкими й орієнтованими на бюрократичний ручний аудит, або занадто специфічними й обмеженими рамками конкретної індустрії. Сліпе копіювання будь-якого з цих підходів неминуче призвело б до критичного зниження гнучкості системи, надлишковості обчислювальних алгоритмів та ускладнення архітектури самого сканера. Саме тому жодна з перелічених методологій не була обрана як єдина чи базова основа для реалізації практичної частини дослідження.

Така невідповідність зумовлена тим, що класичні стандарти створювалися для «статичного» світу паперової сертифікації, тоді як автоматизований аналіз

APK-файлів вимагає максимальної швидкості, адаптивності та здатності працювати в умовах безперервної інтеграції. Спроба автоматизувати, наприклад, вимоги Common Criteria вимагала б створення складних систем експертного оцінювання, які б генерували величезну кількість хибнопозитивних спрацьованого і потребували б постійного втручання людини. З іншого боку, орієнтація виключно на OWASP MASVS, попри її технічну привабливість, суттєво обмежила б аналітичний потенціал сканера, залишаючи поза увагою специфічні поведінкові аномалії додатка або вимоги щодо корпоративних політик безпеки, які чітко прописані в документах NIST.

У зв'язку з цим, у межах цього проекту було прийнято рішення відмовитися від догматичного наслідування одного стандарту на користь розробки власної гібридної, полегшеної моделі оцінювання вразливостей. Цей підхід базується на принципі синергії: з кожної світової методології було виокремлено лише ті елементи, які піддаються чіткій математичній формалізації та автоматизації. Створення такого кастомізованого, модульного фреймворку дозволило зберегти максимальну гнучкість інструменту, забезпечити високу швидкість сканування APK-файлів та адаптувати процес аналізу під конкретні практичні завдання дослідження без зайвого методологічного баласту.

1.4. Технологічний стек та інструментарій автоматизованого сканування

Масштаби та складність сучасного програмного забезпечення для мобільних платформ вимагають використання диверсифікованого підходу до аналізу безпеки. Проведення аудиту виключно в ручному режимі в сучасних реаліях є неефективним та економічно недоцільним через високі часові витрати. Саме тому в індустрії кібербезпеки сформувався потужний ринок автоматизованих рішень, що класифікуються за методологією аналізу на інструменти статичного тестування безпеки, динамічного аналізу, зворотного проектування та спеціалізовані сканери складних архітектурних середовищ. Для побудови ефективного конвеєра оцінювання вразливостей необхідно детально розглянути як базові інструменти, що склали основу даного проекту, так і інші індустріальні рішення, оцінивши їхні переваги, недоліки та актуальність.

Одним із найбільш фундаментальних кроків під час аналізу будь-якого APK-файлу є етап декомпіляції та дезасемблювання, оскільки дослідник має справу з бінарним скомпільованим пакетом. Для вирішення цього завдання в індустрії використовуються кілька ключових рішень:

1. JADX: Потужний декомпілятор, що забезпечує швидке перетворення байт-коду Dalvik у високорівневий і читабельний код на мові Java [22]. Він підтримує як графічний інтерфейс, так і утиліту командного рядка, що робить його ідеальним для інтеграції в автоматизовані скрипти.

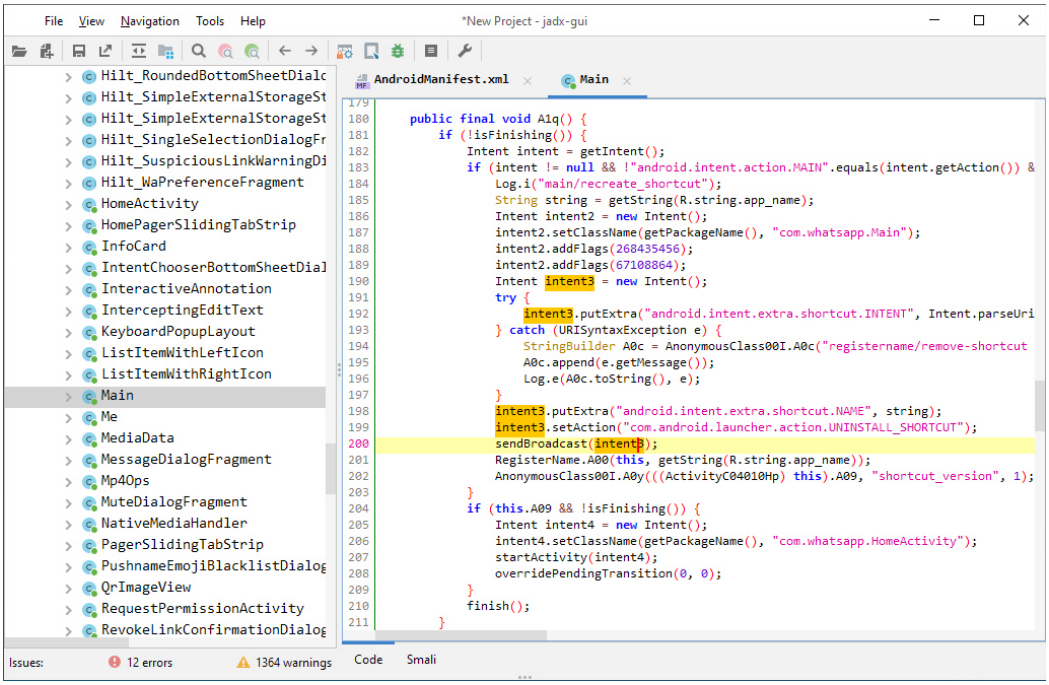


Рисунок 1.3.1 – Використання утиліти JADX для декомпіляції APK-файлу

2. Apktool: Інструмент, орієнтований на розпакування бінарних ресурсів додатка до їхнього початкового стану, а також трансляцію DEX-файлів у проміжний асемблерний код Smali [23].

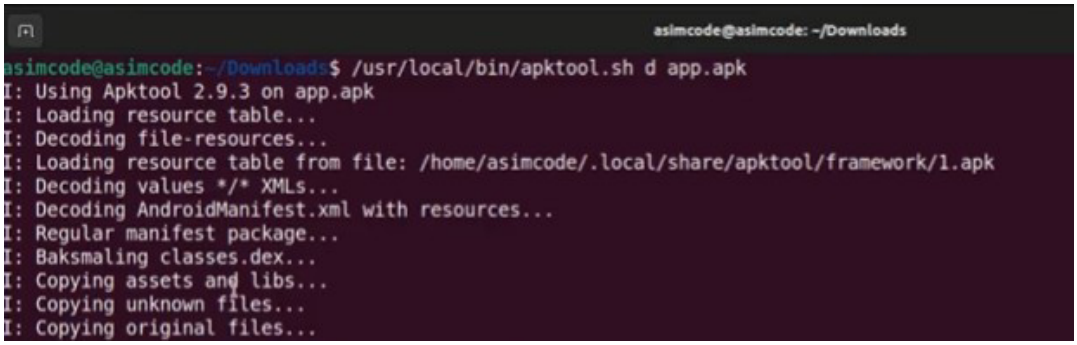


Рис. 1.3.2 – Використання утиліти JADX для дезасемблювання APK-файлу

Після отримання вихідного коду або структури ресурсів постає завдання автоматизованого пошуку вразливостей. Для цього використовуються інструменти семантичного та сигнатурного аналізу вихідних текстів:

1. Semgrep: Утиліта статичного аналізу нового покоління, яка сканує код не як звичайний текст, а будує його абстрактне синтаксичне дерево. Це дозволяє писати правила, які розуміють контекст виконання коду, логіку оголошення змінних та виклики функцій [24].

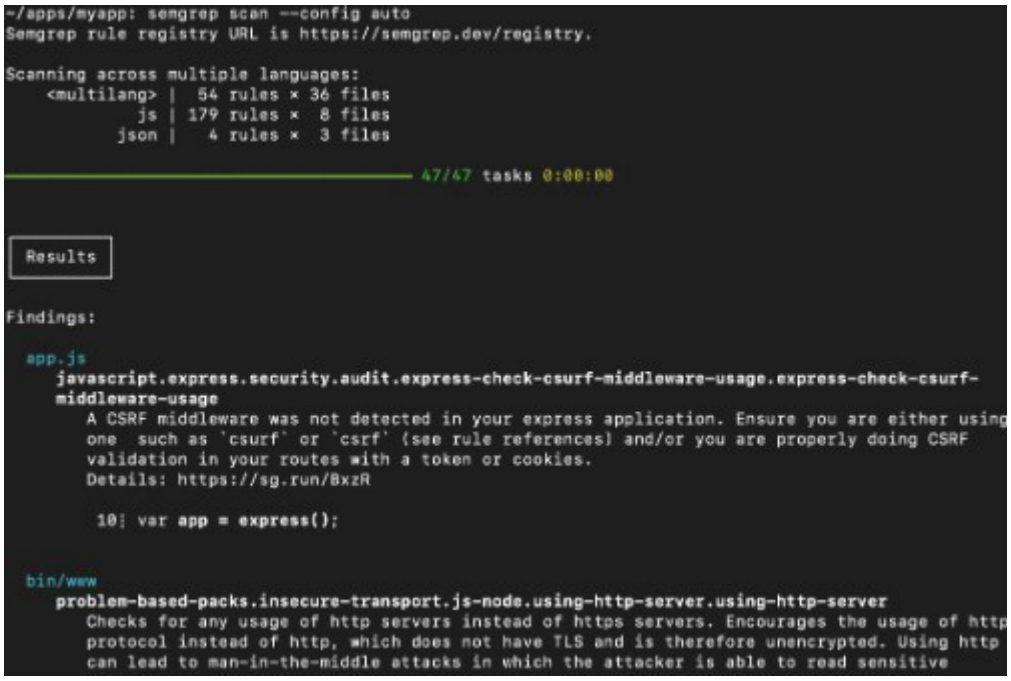


Рисунок 1.3.3 – Приклад використання інструменту semgrep

2. Регулярні вирази: Класичний метод сигнатурного пошуку текстових збігів за заздалегідь визначеними шаблонами.

Для дослідження складних взаємозв'язків даних та обробки специфічних сучасних архітектур технологічний стек доповнюється вузькоспеціалізованим інструментарієм:

1. Mariana Trench: Розроблений компанією Meta інструмент аналізу потоків даних, спеціально спроектований та оптимізований для роботи з великими Android-додатками на рівні байт-коду.

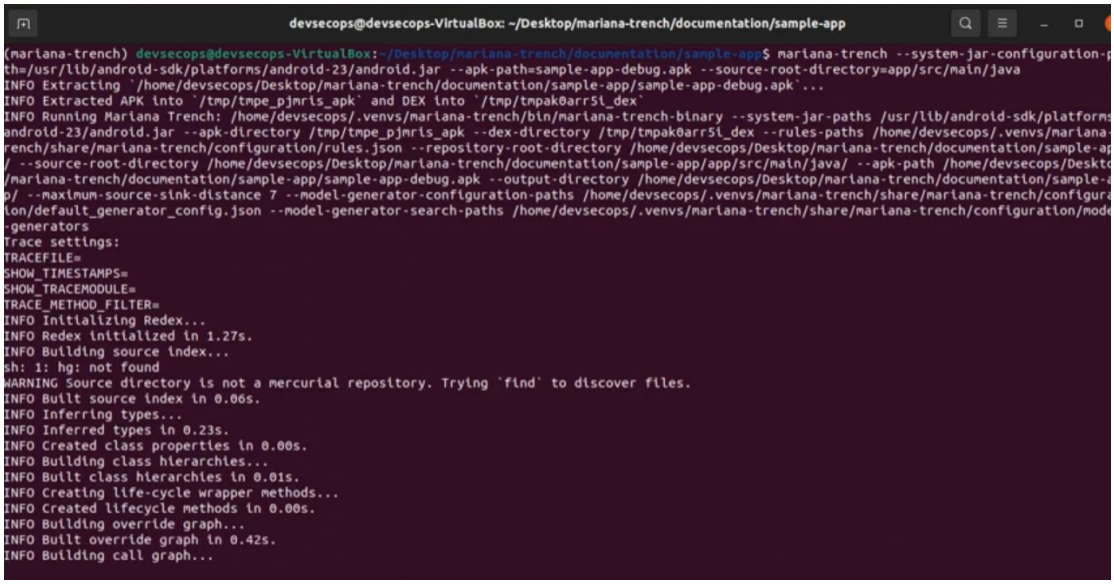


Рисунок 1.3.4 – Приклад використання інструменту Mariana Trench

2. Blutter: Сучасна спеціалізована утиліта для зворотного проектування мобільних додатків, що створені за допомогою кросплатформного фреймворку Flutter. Оскільки Flutter компілює вихідний Dart-код безпосередньо в нативні машинні інструкції, класичні Java-декомпілятори виявляються безсилими.

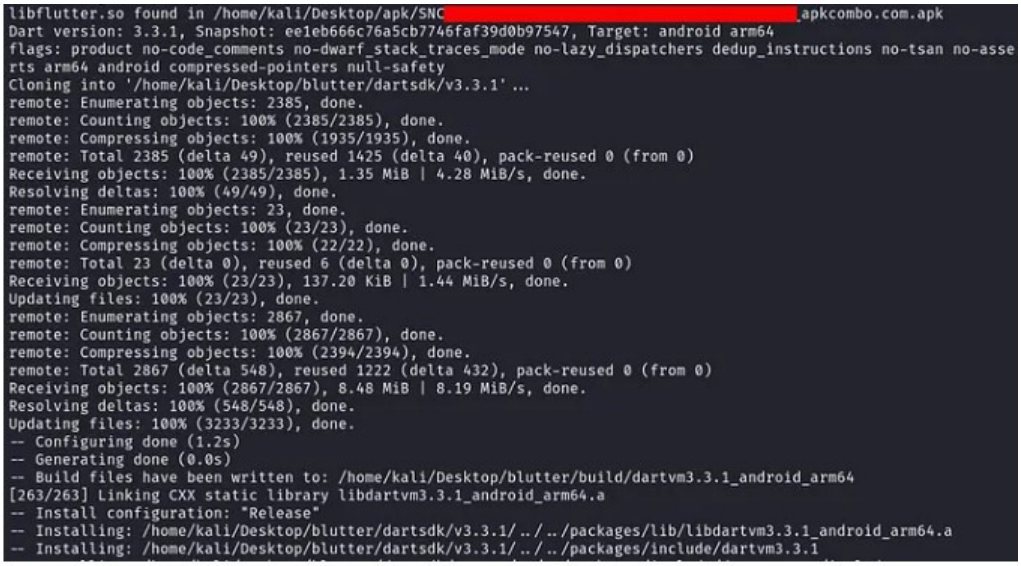


Рисунок 1.3.5 – Приклад використання інструменту blutter

На ринку також присутні інші відомі альтернативні інструменти та автоматизовані платформи, які мають свої унікальні особливості:

1. MobSF: Популярна автоматизована "все-в-одному" платформа з відкритим вихідним кодом, що поєднує в собі інструменти для статичного, динамічного аналізу та тестування API мобільних додатків для Android та iOS [25]. Має великий

недолік в слабкому аналізу коду у порівнянні з глибоким AST-скануванням через Semgrep.

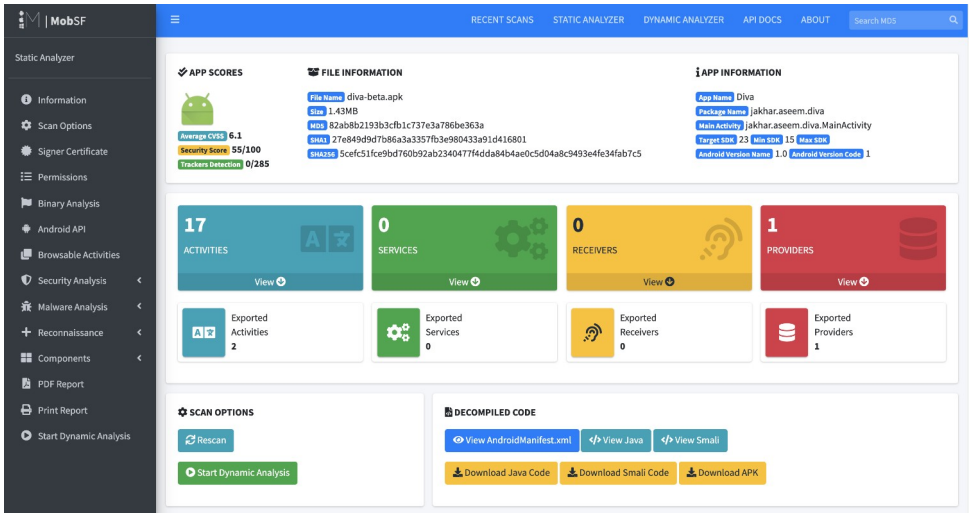


Рисунок 1.3.6 – Використання платформи MobSF для аналізу APK

2. Frida: Потужний та гнучкий інструмент динамічної інструментації,, який завоював статус фактичного стандарту в індустрії аналізу безпеки мобільних додатків. Головна технологічна перевага цього рішення полягає у можливості здійснювати ін'єкції власного програмного коду та виконувати довільні скрипти на мові JavaScript безпосередньо всередині адресного простору та пам'яті працюючого процесу досліджуваного додатка в режимі реального часу [26]. З суттєвих недоліків потребує root-прав, вимагає написання індивідуальних скриптів під кожен конкретний додаток.

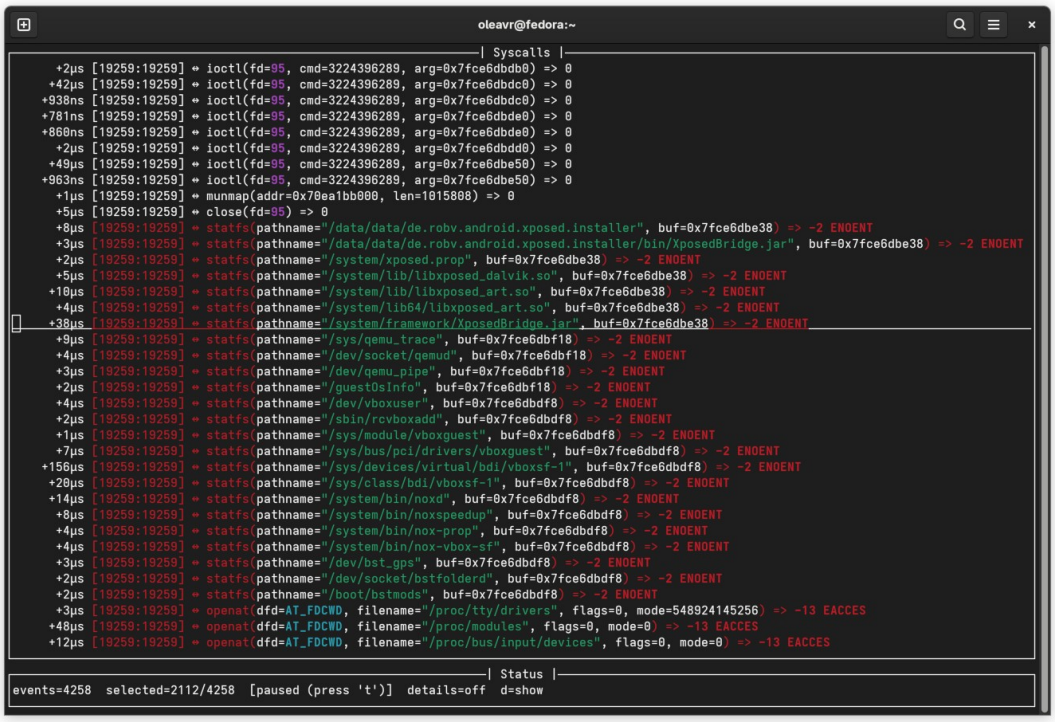


Рисунок 1.3.7 – Використання інструменту Frida для перехоплення викликів

Таким чином, сучасний технологічний стек автоматизованого сканування APK-файлів демонструє перехід від простих сигнатурних методів до складних систем аналізу логіки та адаптації під нові кросплатформні фреймворки, що дозволяє забезпечити максимальну точність та швидкість оцінювання захищеності мобільного софту.

РОЗДІЛ 2. МЕТОДИ СТАТИЧНОГО ТА СЕМАНТИЧНОГО ОЦІНЮВАННЯ ЗАХИЩЕНОСТІ APK-ФАЙЛІВ

2.1. Концептуальний план аналізу бінарних артефактів та архітектура триетапного конвеєра.

Ефективне дослідження захищеності сучасного мобільного програмного забезпечення в умовах обмеженого часового ресурсу та високої щільності релізів вимагає відмови від хаотичного ручного аналізу на користь чітко структурованого, детермінованого та автоматизованого процесу. У рамках сучасних парадигм розробки, таких як Agile та DevSecOps, оновлення мобільних додатків можуть випускатися кілька разів на тиждень, а в деяких випадках — і щодня. За таких умов традиційні підходи до проведення тестування на проникнення, що передбачають залучення високооплачуваних експертів для ручного реверс-інжинірингу, стають головним вузьким місцем усього життєвого циклу розробки. Вони уповільнюють вихід продукту на ринок або призводять до того, що на реліз виходять компрометовані пакети, перевірити які просто не встигли.

Побудова цілісної стратегії безпекового аудиту бінарних пакетів програм базується на послідовному зануренні в архітектурну логіку додатка: від поверхневого вивчення системних конфігурацій до детального математичного моделювання внутрішніх потоків даних і деобфускації бінарного коду. Головна проблема аналізу готового вихідного артефакту полягає в тому, що він є «чорною скринькою». На відміну від аналізу вихідного коду, де інженер бачить чітку структуру та коментарі, бінарний аналіз має справу із заплутаним, оптимізованим компілятором байткодом або машинними інструкціями. Щоб не витрачати надлишкові обчислювальні ресурси на повний аналіз терабайтів бінарних даних, процес необхідно формалізувати у вигляді конвеєрної обробки.

Для реалізації цього підходу розроблено триетапний конвеєр автоматизованого оцінювання захищеності. Він оптимізує використання обчислювальних потужностей шляхом розподілу методів аналізу за рівнем їхньої ресурсомісткості, глибини інтерпретації та швидкості обробки інформації. Кожен етап конвеєра виступає у ролі своєрідного інтелектуального фільтра. Якщо додаток

містить критичні архітектурні дефекти, які можна виявити миттєво, немає сенсу запускати складні та ресурсомісткі алгоритми математичного моделювання коду. Весь процес дослідження розділено на три послідовні стадії, кожна з яких має чітко визначені вхідні дані, математичний апарат та критерії зупинки процесу.

Перша стадія конвеєра, що позначається як Stage 0, є найшвидшою та найменш обчислювально витратною, проте вона закладає фундамент для всього подальшого аналізу. На цьому етапі об'єкт дослідження – APK-пакет – розглядається не як сукупність виконуваного коду, а як структурований архів, що містить метадані, конфігураційні константи та декларативні описові структури. Основна мета Stage 0 полягає в проведенні миттєвої верифікації базових параметрів безпеки платформи Android, без виконання декомпіляції самого байткоду.

Процес починається з автоматизованого вилучення та парсингу головного конфігураційного документа додатка – файлу AndroidManifest.xml. Оскільки в самому APK-файлі цей документ зберігається у бінарному форматі Android Binary XML, конвеєр використовує швидкісні лексери та парсери для його трансляції у стандартний деревоподібний вигляд XML. Після цього запускається серія експрес-перевірок за такими векторами:

1. Аналіз конфігурації компонентів: Програма сканує всі задекларовані сутності – Activities, Services, Broadcast Receivers та Content Providers. Особлива увага приділяється прапорцю android:exported. Якщо компонент є експортованим, це означає, що будь-який інший сторонній додаток, встановлений на пристрої користувача, може надіслати йому міжпроцесний запит (Intent) і змусити виконати внутрішню логіку. Конвеєр автоматично фіксує площу атаки додатка.

2. Аудит дозволів: Перевіряється список затребуваних дозволів. Роботизована система зіставляє цей список із призначенням додатка. Наявність надлишкових дозволів, таких як доступ до SMS, геопозиції чи камери для простого калькулятора або текстового редактора, автоматично підвищує індекс ризику.

3. Перевірка налагоджувальних прапорців: Сканується наявність параметрів android:debuggable та android:allowBackup. Якщо розробник забув вимкнути режим

налагодження перед релізом, зловмисник зможе підключити дебагер до процесу програми та перехопити керування пам’яттю.

Паралельно з аналізом маніфесту на Stage 0 виконується критично важлива процедура – валідація схем цифрового підпису. В архітектурі Android безпека тримається на моделі автора підпису: операційна система не дозволить встановити додаток або його оновлення, якщо підпис скомпрометовано. Конвеєр аналізує вміст директорії META-INF/ та перевіряє, які саме версії схем підпису були використані.

Відсутність підтримки сучасних схем робить додаток вразливим до атак типу Janus, коли зловмисник може додати шкідливі байткодові блоки в APK-архів без порушення цілісності підпису. Експрес-парсинг триває лічені секунди, але дозволяє відсіяти пошкоджені, підроблені або очевидно небезпечні збірки ПЗ.

Якщо додаток успішно пройшов первинні фільтри Stage 0, він переходить на наступний рівень конвеєра – Stage 1. Цей етап характеризується вищою обчислювальною складністю, оскільки система переходить від аналізу статичних конфігураційних файлів до дослідження безпосередньо логіки додатка, скомпільованої у виконувані файли classes.dex.

Головне завдання Stage 1 – перетворення низькорівневого байткоду віртуальної машини Dalvik/ART у людиночитаний або придатний для аналізу формат з подальшим застосуванням методів лінійного сигнатурного пошуку. Для цього конвеєр ініціює процес автоматизованої декомпіляції. Використовуючи оптимізовані декомпілятори, система розгортає внутрішню структуру додатка, відновлюючи назви класів, методів та інтерфейсів.

Після того як текстове представлення коду або його проміжне представлення сформовано, запускається механізм лінійного сигнатурного аналізу. Цей метод базується на скануванні тексту програми за допомогою заздалегідь визначеної бази регулярних виразів, сигнатур вразливостей та правил. Перевага лінійного аналізу полягає у його високій швидкості обробки великих масивів даних. На цьому етапі конвеєр шукає:

1. Захардкоджені конфіденційні дані: Шукаються текстові рядки, які за шаблонами схожі на приватні ключі шифрування, API-токени сервісів, паролі доступу до баз даних або тестові облікові записи, залишені розробниками під час відладки.

2. Використання небезпечних API: Система здійснює пошук викликів функцій, які за своєю природою є ризикованими. Сюди відносяться виклики застарілих криптографічних алгоритмів, які є математично вразливими до колізій та швидкого підбору.

3. Небезпечні налаштування мережевих з'єднань: Пошук у коді викликів, що вимикають перевірку SSL/TLS сертифікатів, наприклад, кастомні класи TrustManager, які приймають будь-які сертифікати. Це прямий шлях до компрометації даних через атаки типу Man-in-the-Middle.

Проте лінійний сигнатурний аналіз має суттєвий методологічний недолік – він повністю ігнорує контекст виконання програми. Текстовий пошук не знає, чи буде викликаний цей шматок коду взагалі, чи передаються туди реальні дані користувача, і чи є цей виклик частиною мертвого коду. Як наслідок, Stage 1 генерує певну кількість хибнопозитивних спрацьовувань. Його мета – сформувати первинну «карту підозрілих місць» додатка для передачі на фінальний, найглибший етап аналізу.

Фінальним і найбільш інтелектуально містким етапом конвеєра є Stage 2. Сюди потрапляють лише ті артефакти або окремі ізольовані класи коду, які були відмічені як підозрілі на попередніх стадіях. Stage 2 повністю відмовляється від лінійного сприйняття тексту програми та переходить до її глибокого семантичного моделювання.

Основою для проведення аналізу на цьому етапі є побудова абстрактного синтаксичного дерева та графів потоку керування. Замість обробки рядків тексту, конвеєр транслює код у складну математичну граф-структуру, де вузлами є оператори, змінні та виклики функцій, а ребрами – логічні зв’язки та послідовність їхнього виконання. Це дозволяє аналізувати не просто синтаксис, а реальну поведінку програми.

Головною технологічною перлиною Stage 2 є впровадження Taint-аналізу. Математична модель Taint-аналізу оперує двома основними поняттями: джерело (Source) та сток/приймач (Sink) [27].

1. Source (Джерело): Це точка входу, через яку в додаток потрапляють неперевірені або чутливі дані користувача (наприклад, поле введення пароля, GPS-координати, зчитувач SMS або дані з мережевого сокета). Конвеєр автоматично «мітить» ці дані як «забруднені» (tainted).

2. Sink (Приймач): Це критична функція, виконання якої з неперевіреними даними може призвести до катастрофічних наслідків.

Завдання системи полягає в трасуванні графа та відстеженні того, чи існує такий логічний шлях виконання програми, за якого мічені дані з джерела потрапляють у приймач без проходження процедури очищення (Sanitization) або валідації. Якщо такий шлях знайдено – конвеєр фіксує підтверджену вразливість із мінімальним рівнем хибних спрацьовувань.

Така триетапна архітектура дозволяє відсікати завідомо дефектні збірки вже на перших секундах роботи конвеєра, поступово нарощуючи аналітичну складність дослідження лише для тих компонентів, які пройшли первинні фільтри безпеки. Розподіл завдань забезпечує високу пропускну здатність системи. Замість того щоб витрачати години обчислювального часу серверів на повний семантичний аналіз кожного файлу в кожному додатку, конвеєр витрачає понад 80% часу лише на ті ділянки коду, які дійсно несуть у собі потенційну загрозу.

Головною перевагою впровадження такого підходу є масштабованість. Конвеєр може бути легко розгорнутий у хмарній інфраструктурі всередині Docker-контейнерів та інтегрований у CI/CD процеси великих розробників мобільного ПЗ. Роботизована система працює автономно: при кожному новому коміті розробника або при збірці нічного релізу конвеєр автоматично підхоплює бінарний артефакт, проводить його через усі три стадії аналізу та видає офіцеру з безпеки чіткий, математично обґрунтований вердикт із зазначенням конкретних вразливостей, рівнів їхнього ризику та рекомендацій щодо їх усунення. Це дозволяє досягти

безпрецедентного рівня захищеності мобільних додатків без зниження темпів бізнес-розробки.

2.2. Методи експрес-парсингу метаданих та сигнатурного аналізу вихідного коду.

Перші два етапи розробленого автоматизованого конвеєра оцінювання захищеності – Stage 0 та Stage 1 – орієнтовані на швидку обробку вхідного бінарного артефакту з метою виявлення явних архітектурних та конфігураційних дефектів. Оскільки повний аналіз поведінки програми є обчислювально складною задачею, первинна фільтрація за допомогою статичних методів дозволяє оптимізувати навантаження на систему.

Процес аналізу починається з розбору структури APK-файла, який з точки зору архітектури є ZIP-архівом, що містить скомпільований байткод, ресурси, сирі файли та метадані цифрового підпису. Головним елементом конфігурації екосистеми Android є файл AndroidManifest.xml. Проте, у фінальній збірці додатка цей файл зберігається не у текстовому форматі, а у вигляді оптимізованого бінарного дерева Android Binary XML.

Для роботизованого експрес-парсингу на Stage 0 застосовується спеціалізований лексер, який зчитує байтовий потік AXML, розшифровує таблицю рядків та відновлює ієрархію тегів без залучення важковагових декомпіляторів. Це дозволяє здійснювати перевірку конфігурації за мілісекунди.

Основна увага під час парсингу маніфесту приділяється детектуванню трьох критичних прапорців, які безпосередньо впливають на площу атаки:

1. android:exported: Цей атрибут визначає доступність компонентів додатка для зовнішніх процесів. Якщо для компонента, що виконує чутливі функції, встановлено значення true, автоматизована система фіксує потенційну вразливість міжкомпонентної взаємодії. Зловмисний додаток, встановлений у системі, зможе надіслати несанкціонований Intent для обходу екранів автентифікації.

2. android:allowBackup: Якщо цей параметр активовано, операційна система дозволяє копіювання приватної пісочниці додатка через інтерфейс налагодження ADB. Автоматизований сканер маркує це як ризик фізичного витоку даних,

оскільки зловмисник, отримавши доступ до розблокованого пристрою, може злити сесійні токени або локальні бази даних.

3. android:debuggable: Активація цього прапорця у релізній збірці дозволяє підключення віртуальної машини до процесу через протокол JDWP. Конвеєр автоматично відхиляє такі збірки, оскільки режим налагодження повністю нівелює захисні механізми ОС, дозволяючи зчитувати пам'ять додатка в реальному часі.

Паралельно з аналізом конфігурації, Stage 0 виконує низькорівневу перевірку цілісності пакету через аудит цифрового підпису в каталозі META-INF/. Еволюція механізмів захисту Android породила чотири покоління схем підпису:

1. v1 (JAR-signing): Перевіряє лише цілісність окремих файлів. Файли маніфесту містять хеш-суми для кожного бінарного артефакту.

2. v2 (APK Signature Scheme v2): Підписує весь бінарний файл цілісно, захищаючи блоки даних між таблицею файлів та кінцем архіву.

3. v3/v4: Додають підтримку ротації сертифікатів та стримінгового завантаження даних.

Автоматизований сканер перевіряє наявність підписів вищих рівнів. Якщо додаток підписаний виключно за схемою v1, система фіксує критичну вразливість до архітектурної атаки Janus. Суть цієї атаки полягає в тому, що формат файлу APK є одночасно валідним ZIP-архівом і валідним файлом DEX, якщо байткод записати на початку файлу. За відсутності захисту схеми v2, зловмисник може впровадити шкідливий кодовий блок на початок пакета. Операційна система Android під час перевірки підпису v1 бачитиме лише чисті файли в архіві, але при запуску віртуальна машина ART почне виконання з лінійного DEX-блоку зловмисника, що призведе до повної компромерації пристрою.

2.3. Семантичний AST-аналіз та моделювання небезпечних потоків даних.

Коли лінійне сканування фіксує потенційні точки ризику, конвеєр активує Stage 2 – рівень глибокого семантичного моделювання. Головна мета цього етапу полягає в тому, щоб зрозуміти не просто синтаксичний склад коду, а його реальну логіку та поведінкові патерни в процесі виконання.

Для переходу від тексту до семантики аналізатор транслює декомпілізований байткод у математичну структуру – Абстрактне Синтаксичне Дерево. AST – це орієнтований деревоподібний граф, де внутрішні вузли представляють оператори мови програмування (виклики методів, розгалуження if-else, цикли, присвоєння змінних), а листки – операнди (імена змінних, константи, літерали).

На основі AST-дерева система будує Граф потоку керування (Control Flow Graph – CFG). Вузол графа CFG представляє собою базовий блок – послідовність операторів, яка має лише одну точку входу та одну точку виходу. Ребра графа вказують на можливі напрямки передачі керування (наприклад, умовні переходи).

Маючи таку математичну модель, аналізатор може абстрагуватися від того, як саме розробник назвав змінні чи класи (що нівелює просту обфускацію імен), і аналізувати топологію графа. Наприклад, використання небезпечного виклику функції, знайденого на Stage 1, перевіряється на досяжність: якщо в графі CFG немає жодного шляху від легітимних точок входу (методів життєвого циклу додатка, таких як onCreate) до базового блоку з цим викликом, система класифікує цю знахідку як мертвий код і автоматично знижує її пріоритет, уникаючи відволікання аналітика.

Основу аналізу небезпечних потоків даних складає Taint-аналіз. Його сутність полягає в математичному моделюванні інформаційних потоків всередині програми з метою відстеження поширення потенційно небезпечної або конфіденційної інформації.

Формально модель Taint-аналізу описується як трійка: T = {Source,Sink,Propagation}

Джерело – множина функцій Android API, через які додаток отримує дані, що потребують захисту або є неперевіреними. Конвеєр автоматично маркує повернені цими методами значення прапорцем TAINTED. До джерел належать:

1. Дані користувача та пристрою: TelephonyManager.getDeviceId(), Location.getLastKnownLocation()

2. Зовнішні інтерфейси: Intent.getStringExtra(), TextInput.getText()

3. Мережеві та файлові потоки: InputStream.read()

Стоки – множина критичних функцій API, виконання яких із забрудненими даними може призвести до порушення конфіденційності чи цілісності. До приймачів належать:

1. Витік даних: Log.d(), SmsManager.sendTextMessage()

2. Виконання команд та запитів: Runtime.getRuntime().exec(), SQLiteDatabase.execSQL()

3. Мережева передача: OutputStream.write()

Правила поширення – набір логічних правил, які визначають, як мітка TAINTED переходить від однієї змінної до іншої. Наприклад, якщо змінна A є міченою, то в результаті виконання операції присвоєння або конкатенації рядків B = A + \text{"_data"}, змінна B також отримує статус TAINTED.

Процес пошуку вразливостей за допомогою Taint-аналізу зводиться до задачі пошуку шляхів на графах. Роботизована система виконує два типи аналізу:

1. Forward Taint Analysis (Прямий аналіз): Рух починається від знайденого джерела. Система іде вперед по графу потоку даних, маркуючи всі залежні змінні, і перевіряє, чи не досягне якийсь із мічених потоків функції-приймача.

2. Backward Taint Analysis (Зворотний аналіз): Рух починається від приймача. Система аналізує аргументи функції та рухається назад по графу, намагаючись з'ясувати походження цих даних і перевіряючи, чи не прийшли вони з неперевіреного джерела.

Під час обходу графа алгоритм враховує наявність так узваних санітайзерів — функцій очищення або валідації даних. Якщо на шляху між джерелом і приймачем мічена змінна передається як аргумент у функцію, яка виконує, наприклад, екранування символів для SQL-запитів або шифрування перед логуванням, мітка TAINTED знімається.

Вразливість фіксується конвеєром лише тоді, коли існує хоча б один транзитний шлях на графі CFG від джерела до стоку, де дані не пройшли через жоден легітимний санітайзер. Наприклад, якщо ідентифікатор пристрою IMEI (мічений Source) безпосередньо передається у виклик системного логера Log.w() –

система реєструє стовідсоткову вразливість витоку персональних даних (CWE-200: Information Exposure) і автоматично формує доказову базу для фінального звіту.

Застосування семантичного аналізу та моделювання потоків даних на Stage 2 дозволяє кардинально знизити рівень хибних спрацьовувань автоматизованого конвеєра, забезпечуючи високу точність оцінювання захищеності APK-файлів.

2.4. Поглиблений семантичний AST-аналіз та механізми інструментального трасування ланцюжків викликів

Останній, найбільш технологічно складний етап архітектури конвеєра орієнтований на повне розкриття внутрішньої логіки декомпільованого додатка та верифікацію його реальних поведінкових патернів. Якщо попередні стадії конвеєра працювали з декларативними метаданими та лінійним текстовим масивом регуляцій, то на поточному етапі дослідження переходить у площину математичного моделювання графів та семантичного розбору складних синтаксичних структур.

Головна науково-практична проблема автоматизованого аналізу великих програмних комплексів полягає у високому рівні помилок другого роду — хибнопозитивних спрацьовувань, які виникають через брак контексту при лінійному скануванні. Для досягнення максимальної глибини аналізу та мінімізації показника інформаційного шуму утиліти автоматизації оперують двома фундаментальними технологіями: семантичним скануванням за допомогою інструменту Semgrep та міжмодульним глобальним трасуванням.

Фундаментальним недоліком класичних засобів статичного аналізу є їхня чутливість до найменших змін у стилі написання коду або синтаксичній структурі програми. Додавання зайвого пропуску, перенесення рядка, використання альтернативних мовних конструкцій або банальне перейменування локальної змінної повністю нівелює ефективність лінійних сканерів, призводячи до пропуску критичних уразливостей. Для подолання цього ліміту та забезпечення інваріантності аналізу щодо синтаксичного шуму використовують інструмент Semgrep.

В основі роботи Semgrep лежить алгоритм перетворення плоского тексту вихідного коду мов Java або Kotlin у абстрактне синтаксичне дерево. AST являє собою кінцевий орієнтований граф, у якому внутрішні вершини зіставляються з операторами мови програмування, а листя – з відповідними операндами. Математична формалізація аналізу в Semgrep базується на зіставленні графів. Замість пошуку текстових збігів, рушій Semgrep транслює аналітичне правило безпеки у абстрактний шаблон коду, який згодом компілюється у фрагмент синтаксичного дерева. Пошук уразливостей зводиться до ізоморфного мапування шаблону на глобальне AST-дерево декомпільованого APK-файла.

Особливою перевагою Semgrep у конвеєрі є використання метазмінних та оператора трикрапки. Метазмінні (наприклад, $X, $USER) дозволяють фіксувати абстрактні ідентифікатори, назви яких заздалегідь невідомі, забезпечуючи відстеження їхньої тотожності всередині області видимості. Оператор трикрапки дозволяє ігнорувати надлишкові або проміжні інструкції коду, які не впливають на суть уразливості.

Наприклад, синтаксичний шаблон перевірки ініціалізації небезпечного криптографічного шифру в Java може мати вигляд:

```
pattern: Cipher.getInstance("$ALGO", ...)
```

Рисунок 2.4.1 – Приклад синтаксичного шаблону semgrep для Java

Рушій Semgrep успішно локалізує цей виклик у AST-дереві незалежно від того, чи передаються додаткові параметри провайдера безпеки, чи розбитий цей виклик на кілька рядків, і які саме назви мають проміжні змінні. Це дозволяє здійснювати швидкісну семантичну фільтрацію кодової бази на наявність структурних дефектів.

Самого лише факту наявності вразливої або небезпечної конструкції в коді, зафіксованого інструментом Semgrep, недостатньо для винесення остаточного автоматизованого вердикту про критичність ризику. У великих програмних комплексах або при широкому використанні сторонніх open-source бібліотек та SDK (наприклад, залишків тестових модулів або аналітичних фреймворків) уразливий метод може перебувати в абсолютно ізольованому стані. Такий стан

класифікується як «мертвий код» - ділянка програми, яка за жодних обставин не може бути викликана під час реального виконання програми користувачем чи операційною системою. Якщо шлях виконання від зовнішніх граней додатка до вразливої функції відсутній, така уразливість не несе реальної загрози і не може бути експлуатована зловмисником.

Повідомлення про наявність дефектів у мертвому коді створює критичний обсяг помилкових тривог, що паралізує роботу аналітиків безпеки. Для доведення експлуатабельності виявлених дефектів та математичного підтвердження наявності інформаційних маршрутів конвеєр реалізує процедуру глобального трасування ланцюжків викликів. Це завдання вимагає побудови міжмодульного графу викликів, де вершинами є окремі методи всіх класів додатка, а ребра показують відношення виклику («метод А викликає метод Б»). Побудова такого графу ускладнюється масштабністю мобільних додатків, де кількість методів може сягати сотень тисяч, що вимагає залучення дворівневої системи оптимізації обчислень.

Процес трасування та побудови графу викликів у межах розробленого конвеєра виконується за чітко детермінованим алгоритмом, який поєднує надшвидку низькорівневу фільтрацію та глибокий глобальний аналіз потоків даних. Весь процес розділено на 3 послідовні кроки.

На першому підетапі конвеєр намагається уникнути витратного повного розгортання всього об'єму байт-коду. Інструмент dexfinder здійснює швидке низькорівневе сканування структури індексів компонентів безпосередньо у бінарних файлах classes.dex. Робота утиліти базується на читанні заголовків файлу DEX та парсингу специфічних внутрішніх таблиць: string_ids, type_ids, proto_ids та, найголовніше, method_ids.

На другому підетапі – його завдання миттєво локалізувати всі точки згадування (строкові референси та індекси викликів) цільових класів, методів та інтерфейсів, які Semgrep позначив як потенційно небезпечні. dexfinder формує первинну матрицю суміжності для тих ділянок бінарного файлу, які фізично містять посилання на дефекти. Це дозволяє відсікти до 70% завідомо «чистих» класів додатка від подальшого обходу.

Для того щоб прокласти маршрут виконання програми, необхідно чітко визначити легітимні початкові точки, з яких операційна система Android або сторонні процеси можуть розпочати взаємодію з додатком. Як точки входу конвеєр маркує компоненти мобільного додатка, які були ідентифіковані на етапі Stage 0 як експортовані (android:exported="true"), або які містять активні фільтри намірів (<intent-filter>).

Додатково як Entry Points реєструються стандартні системні методи життєвого циклу Android (такі як Activity.onCreate(), Service.onStartCommand(), BroadcastReceiver.onReceive()), які викликаються автоматично у відповідь на зовнішні події. Саме ці компоненти є публічно доступними вікнами для зовнішнього впливу з боку операційної системи або сторонніх шкідливих додатків, встановлених на пристрої.

Після локалізації точок входу та первинного відсікання коду через dexfinder, конвеєр запускає аналітичний рушій. Завдання — реалізувати математично точний міжпроцедурний аналіз потоків даних та побудувати наскрізний граф викликів. Процес реверс-трасування починається від знайдених точок входу та рухається вглиб архітектури додатка, паралельно з прямою побудовою ланцюжків від джерел вразливостей. Інструмент послідовно фіксує, які методи викликають поточний компонент, формуючи безперервний ланцюжок референсів. Математичний апарат оперує концепцією формальних інваріантів для кожного методу. Замість того щоб щоразу переобчислювати внутрішній граф функції при кожному її згадуванні, рушій один раз будує абстрактну модель поведінки методу і використовує це резюме при побудові глобального графу. Це дозволяє конвеєру ефективно обробляти складні рекурсивні виклики та багаторівневу вкладеність методів, яка характерна для сучасних об'єктно-орієнтованих архітектур.

Якщо в результаті аналізу глобальний граф замикається і підтверджує наявність наскрізного, логічно несуперечливого ланцюжка викликів від публічно доступного інтерфейсу IPC (Intent, експортована Activity) безпосередньо до вразливої ділянки коду (наприклад, незахищеної транзакції бази даних або виклику

слабкого криптоалгоритму), уразливість маркується найвищим пріоритетом критичності і вважається реально експлуатованою.

Такий підхід дозволяє повністю автоматизувати процес верифікації, відсікаючи потенційний інформаційний шум та надаючи аудитору безпеки не просто сухий опис дефекту, а готовий вектор для проведення цілеспрямованого тесту на проникнення. Конвеєр автоматично генерує карту маршруту викликів, яка покроково демонструє розробнику, через які саме класи та методи зовнішні дані можуть скомпрометувати безпеку системи. Це забезпечує високу технологічну насиченість конвеєра, дозволяючи впроваджувати його в безперервні процеси розробки комерційного мобільного ПЗ, де критично важливо отримувати миттєвий та точний аналітичний зріз безпеки без залучення ручного реверс-інжинірингу.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ АВТОМАТИЗОВАНОГО КОНВЕЄРА ТА ВЕРИФІКАЦІЯ РЕЗУЛЬТАТІВ ОЦІНЮВАННЯ APK-ФАЙЛІВ

3.1 Постановка завдання та вибір вразливого додатку

У межах виконання даної випускної кваліфікаційної роботи ключовим практичним завданням є проектування, програмна реалізація, тестування та валідація інтегрованого автоматизованого конвеєра для оцінювання вразливостей і дефектів безпеки у бінарних файлах мобільних додатків операційної системи Android. Актуальність автоматизації цього процесу зумовлена необхідністю швидкого, масштабованого та точного виявлення конфігураційних помилок і вразливостей коду на етапі перевірки програмного забезпечення перед його публікацією у відкритих репозиторіях

Для того щоб всебічно протезувати працездатність розробленого конвеєра аналізу додатків, перевірити точність сформованих баз правил, а також мінімізувати рівень хибнопозитивних спрацьовувань, виникла об'єктивна потреба у виборі спеціалізованого тестового APK. Дослідження реальних комерційних додатків на початкових етапах тестування пайплайну ускладнюється тим, що наявність засобів комерційного захисту коду та обфускації може приховати базові логічні помилки, заважаючи оцінити чисту ефективність інструментів аналізу. Саме тому як базовий об'єкт практичного дослідження було обрано спеціалізований демонстраційний мобільний додаток із відкритим вихідним кодом – AndroGoat [28].

Додаток AndroGoat розроблений спеціально для фахівців із кібербезпеки, аналітиків шкідливого програмного забезпечення та пентестерів як навмисно вразливе середовище. На відміну від більшості стандартних програмних продуктів, AndroGoat акумулює в собі широкий спектр критичних уразливостей, які охоплюють майже всі категорії сучасного міжнародного стандарту безпеки мобільних платформ OWASP Mobile Top 10. Це робить його ідеальним полігоном для перевірки глибини сканування автоматизованих систем, оскільки додаток містить реальні приклади некоректного збереження даних, незахищеної

міжпроцесної взаємодії, слабкої криптографії та вразливих конфігурацій середовища виконання.

У зв'язку з цим, загальна постановка завдання практичної частини роботи передбачає послідовне розгортання та опис інструментального конвеєра, який здатний в автоматичному режимі прийняти на вхід бінарний файл AndroGoat.apk, провести його крізь усі заплановані стадії аналізу та згенерувати фінальний структурований звіт. Практичний опис реалізації цього завдання логічно розподіляється на наступні взаємопов'язані кроки:

1. Програмна архітектура пайплайну: Розробка керуючих скриптів автоматизації, налаштування середовища виконання, оркестрація паралельної роботи декомпіляторів та сканерів коду для оптимізації швидкодії системи.

2. Експрес-парсинг бінарного пакета (Stage 0): Миттєве вилучення метаданих, перевірка схем цифрових підписів додатка на предмет стійкості до модифікацій та аналіз бінарного файлу маніфесту для локалізації конфігураційних ризиків високого рівня.

3. Паралельна декомпіляція та розпакування (Stage 1): Одночасний запуск утиліт зворотного проектування у багатопотоковому режимі для повного відновлення вихідних текстів програмного продукту та його ресурсів.

4. Багаторівневий статичний аналіз (Stage 2): Сканування отриманого масиву коду за допомогою сигнатурного пошуку для виявлення захардкодженних секретів, та глибокий семантичний AST-аналіз для виявлення вразливих криптографічних примітивів і помилок логіки коду додатка.

5. Моделювання небезпечних потоків даних та трасування: Інтеграція інструментів Taint-аналізу для відстеження маршрутів переміщення інформації від джерел введення до критичних системних методів, побудова ланцюжків викликів та консолідація результатів у фінальний аналітичний звіт.

3.2 Створення скелету пайплайна та реалізація експрес-парсингу метаданих

Першим етапом аналізу APK у даній роботі є експерс-парсинг метаданих. Для реалізації усього коду було обрано мову Python через те, що більшість використаних інструментів написані саме нею. Сам код розташован у теці src а усі

використані інструменти та SDK перебувають у сусідній теці tools що дає змогу розгорнути пайплайн будь де з мінімальними зусиллями. Вхідною точкою програми пайплайну буде файл main.py який буде запускати кожний етап та парсити передані аргументи.

```python
import argparse
import sys
import time
from pathlib import Path

from utils import reset_directory, save_metadata
from stage0 import run_stage_0
from stage1 import run_stage_1
from stage2 import run_stage_2


def main():
    parser = argparse.ArgumentParser(
        description="Android Automated Static Analysis Pipeline (Stage 0 + S
    )
    parser.add_argument(
        "--apk",
        required=True,
        help="Path to the target APK file"
    )
    parser.add_argument(
        "--workdir",
        required=True,
        help="Absolute path to the working directory"
    )

    args = parser.parse_args()

    apk_path = Path(args.apk)
    if not apk_path.exists():
        print(f"[-] Error: APK file not found at {args.apk}")
        sys.exit(1)

    work_dir = Path(args.workdir)
    out_dir = work_dir / "output_report"

    print("=" * 60)
    print("=== STARTING AUTOMATED APK ANALYSIS PIPELINE ===")
    print("=" * 60)

    total_start_time = time.time()
    reset_directory(out_dir)
    # ------------------------------------------------------------
    # STAGE 0:
    # ------------------------------------------------------------
    print("[*] Launching Stage 0: Extracting APK Metadata...")
    try:
        metadata = run_stage_0(str(apk_path))
        out_dir.mkdir(parents=True, exist_ok=True)
        save_metadata(metadata, out_dir)
```

Рисунок 3.2.1 – Python-скрипт головного файлу пайплайну

```
    print("[*] Launching Stage 0: Extracting APK Metadata...")
    try:
        metadata = run_stage_0(str(apk_path))
        out_dir.mkdir(parents=True, exist_ok=True)
        save_metadata(metadata, out_dir)
    except Exception as e:
        print(f"[-] Critical Error in Stage 0: {e}")
        sys.exit(1)

    print("-" * 60)

    # ------------------------------------------------------------
    # STAGE 1:
    # ------------------------------------------------------------
    print("[*] Launching Stage 1: Decompilation & Apk Scanning...")
    try:
        run_stage_1(str(apk_path), str(work_dir), Path(out_dir))
    except Exception as e:
        print(f"[-] Critical Error in Stage 1: {e}")
        sys.exit(1)

    print("-" * 60)
    # ------------------------------------------------------------
    # STAGE 2:
    # ------------------------------------------------------------
    print("[*] Launching Stage 2: Static Analysis...")
    jadx_sources = f"{work_dir}/output_report/jadax_out/"
    try:
        run_stage_2(str(apk_path), str(work_dir), Path(out_dir), jadx_sources)
    except Exception as e:
        print(f"[-] Critical Error in Stage 2: {e}")
        sys.exit(1)

    print("=" * 60)
    print(f"[+] Pipeline finished in {time.time() - total_start_time:.2f}s")
    print(f"[+] All reports are located in: {out_dir}")
    print("=" * 60)



if __name__ == "__main__":
    main()
```

Рисунок 3.2.2 – Python-скрипт головного файлу пайплайну

Також пайплайн буде кожен раз створювати, якщо не була створена та очищати вихідну теку. Для цього було написано допоміжну функцію reset_directory у файлі utils.py.

```python
import shutil
from pathlib import Path
import json


def save_metadata(metadata, out_dir):
    print("\n[+] Stage 0 Metadata Results:")
    print(json.dumps(metadata, indent=4, ensure_ascii=False))

    try:
        report_file = Path(out_dir) / "report_metadata.json"
        with open(report_file, "w", encoding="utf-8") as f:
            json.dump(metadata, f, indent=4, ensure_ascii=False)
        print(f"[+] Metadata successfully saved to: {report_file}")
    except Exception as e:
        print(f"[-] Warning: Could not save metadata file: {e}")

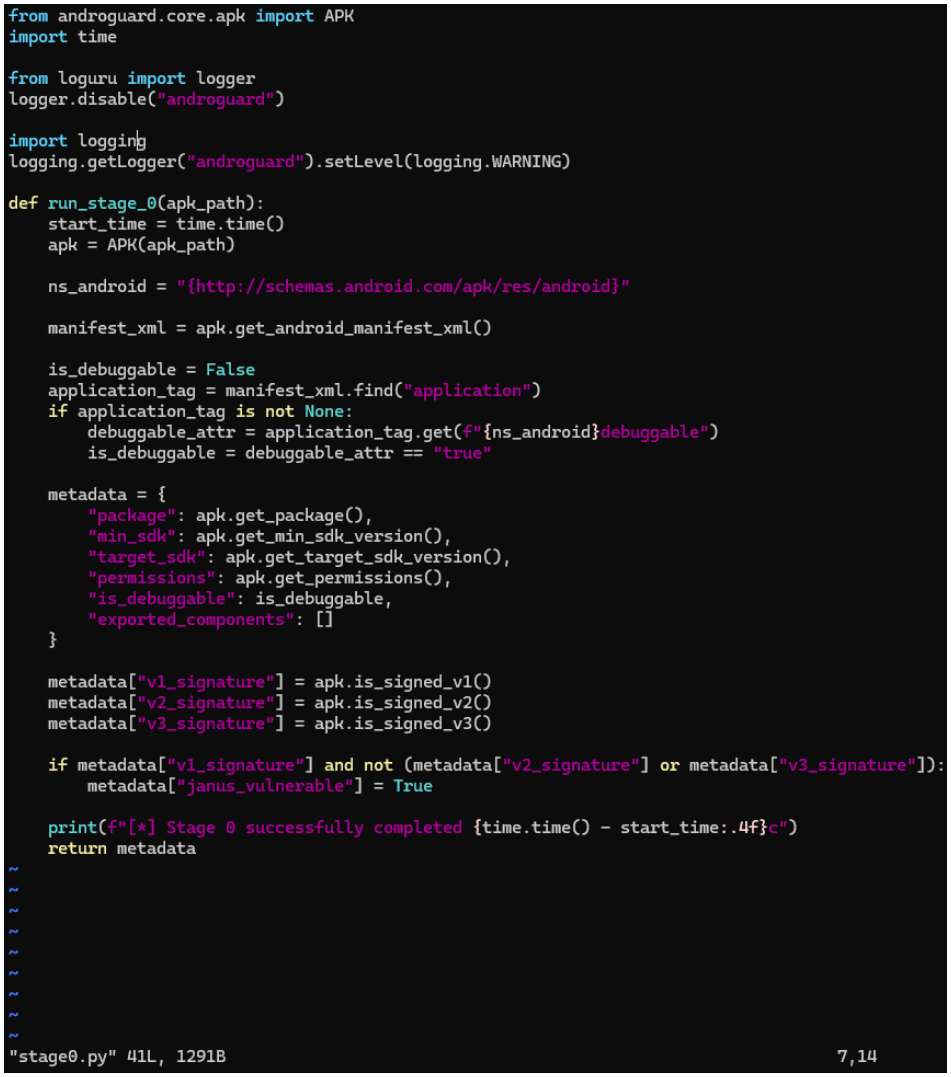

def reset_directory(dir_path: str | Path):
    path = Path(dir_path)

    if path.exists() and path.is_dir():
        shutil.rmtree(path)

    path.mkdir(parents=True, exist_ok=True)
```

Рисунок 3.2.3 – Вспоміжний скрит utils.py

Файл першого етапу має назву stage0.py. Він виконує первинний статичний аналіз APK-файла і використовує для розбору Android-маніфесту бібліотеку androguard [29]. У межах функції run_stage_0 відбувається ініціалізація об'єкта додатка, вилучення маніфесту та парсинг тегу <application> для визначення прапорця is_debuggable. Скрипт збирає базовий словник метаданих, що містить назву пакета, версії SDK, перелік дозволів, а також здійснює послідовну перевірку наявності криптографічних підписів версій v1, v2 та v3. Наприкінці виконання виводиться повідомлення про успішне завершення нульового етапу пайплайну із точним розрахунком витраченого часу, а сформована структура метаданих повертається для подальшої обробки. Результатом цього етапу буде вивід у стандартний потік вводу-виводу та у файл report_metadata.json у вихідній теці. Аргументами пайплайну є apk який аналізується, та робоча тека.



```python
from androguard.core.apk import APK
import time

from loguru import logger
logger.disable("androguard")

import logging
logging.getLogger("androguard").setLevel(logging.WARNING)

def run_stage_0(apk_path):
    start_time = time.time()
    apk = APK(apk_path)

    ns_android = "{http://schemas.android.com/apk/res/android}"

    manifest_xml = apk.get_android_manifest_xml()

    is_debuggable = False
    application_tag = manifest_xml.find("application")
    if application_tag is not None:
        debuggable_attr = application_tag.get(f"{ns_android}debuggable")
        is_debuggable = debuggable_attr == "true"

    metadata = {
        "package": apk.get_package(),
        "min_sdk": apk.get_min_sdk_version(),
        "target_sdk": apk.get_target_sdk_version(),
        "permissions": apk.get_permissions(),
        "is_debuggable": is_debuggable,
        "exported_components": []
    }

    metadata["v1_signature"] = apk.is_signed_v1()
    metadata["v2_signature"] = apk.is_signed_v2()
    metadata["v3_signature"] = apk.is_signed_v3()

    if metadata["v1_signature"] and not (metadata["v2_signature"] or metadata["v3_signature"]):
        metadata["janus_vulnerable"] = True

    print(f"[*] Stage 0 successfully completed {time.time() - start_time:.4f}c")
    return metadata
~
~
~
~
~
~
~
~
~
"stage0.py" 41L, 1291B                                                    7,14
```

Рисунок 3.2.4 – Python-скрипт першого етапу

```
(venv) mor@device323:/mnt/c/Users/Mor/diploma/src/android_pipeline$ python3 main.py --apk /mnt/c/Users
/Mor/diploma/resources/AndroGoat.apk --workdir /mnt/c/Users/Mor/diploma
============================================================
=== STARTING AUTOMATED APK ANALYSIS PIPELINE ===
============================================================
[*] Launching Stage 0: Extracting APK Metadata...
[*] Stage 0 successfully completed 0.1523c

[+] Stage 0 Metadata Results:
{
    "package": "owasp.sat.agoat",
    "min_sdk": "19",
    "target_sdk": "33",
    "permissions": [
        "android.permission.INTERNET",
        "android.permission.USE_BIOMETRIC",
        "owasp.sat.agoat.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION",
        "android.permission.CAMERA",
        "android.permission.USE_FINGERPRINT",
        "android.permission.READ_EXTERNAL_STORAGE",
        "android.permission.WRITE_EXTERNAL_STORAGE"
    ],
    "is_debuggable": true,
    "exported_components": [],
    "v1_signature": true,
    "v2_signature": true,
    "v3_signature": false
}
```

Рисунок 3.2.5 – Запуск пайплайну та результат роботи першого етапу

3.3 Паралельна декомпіляція та аналіз потоків даних

Після фіксації первинних ризиків конвеєр автоматично ініціює паралельний запуск утиліт JADX та apktool. Протягом короткого проміжку відновлюється вихідний Java-код додатка, розпаковуються ресурси. Результат роботи декомпілятора jadx є дуже важливим адже він буде використовуватися у усіх наступних етапах. Вихідний байткод з apktool натомість хоч і не використовується але в деяких випадках буде дуже зручно відкрити його у Android Studio та віддебажити деякі моменти.

```python
def run_jadx(apk_path, out_dir, work_dir):
    cmd = f"{work_dir}/tools/jadx/bin/jadx -d {out_dir} -j 8 {apk_path}"
    subprocess.run(cmd, shell=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

def run_apktool(apk_path, out_dir):
    cmd = f"apktool d {apk_path} -o {out_dir} -f"
    subprocess.run(cmd, shell=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
```

Рисунок 3.3.1 – Реалізація запуску `JADX` та `apktool`

Після відпрацювання цих двох інструментів починає свою роботу аналізатор потоків даних - mariana trench. Для роботи mariana trench потребує дуже багато конфігураційних даних. Потрібно вказати і правила, і моделі, і шлях до SDK з яким має запускатись APK, і, звісно, шлях до ісходного коду.

```
(venv) mor@device323:/mnt/c/Users/Mor/diploma/src/android_pipeline$ cat ../../tools/mariana_trench/con
figuration/lifecycles.json
[
  {
    "base_class_name": "Landroidx/fragment/app/FragmentActivity;",
    "method_name": "fragment_activity_lifecycle_wrapper",
    "callees": [
      {
        "method_name": "onCreate",
        "return_type": "V",
        "argument_types": [
          "Landroid/os/Bundle;"
        ]
      },
      {
        "method_name": "onStart",
        "return_type": "V",
        "argument_types": []
      },
      {
        "method_name": "onResume",
        "return_type": "V",
        "argument_types": []
      },
      {
        "method_name": "onPause",
        "return_type": "V",
        "argument_types": []
      },
      {
        "method_name": "onStop",
        "return_type": "V",
        "argument_types": []
      },
      {
        "method_name": "onRestart",
        "return_type": "V",
        "argument_types": []
      },
      {
        "method_name": "onDestroy",
        "return_type": "V",
        "argument_types": []
      }
```

Рисунок 3.3.2 – Фрагмент файлу конфігурації lifecycles.json

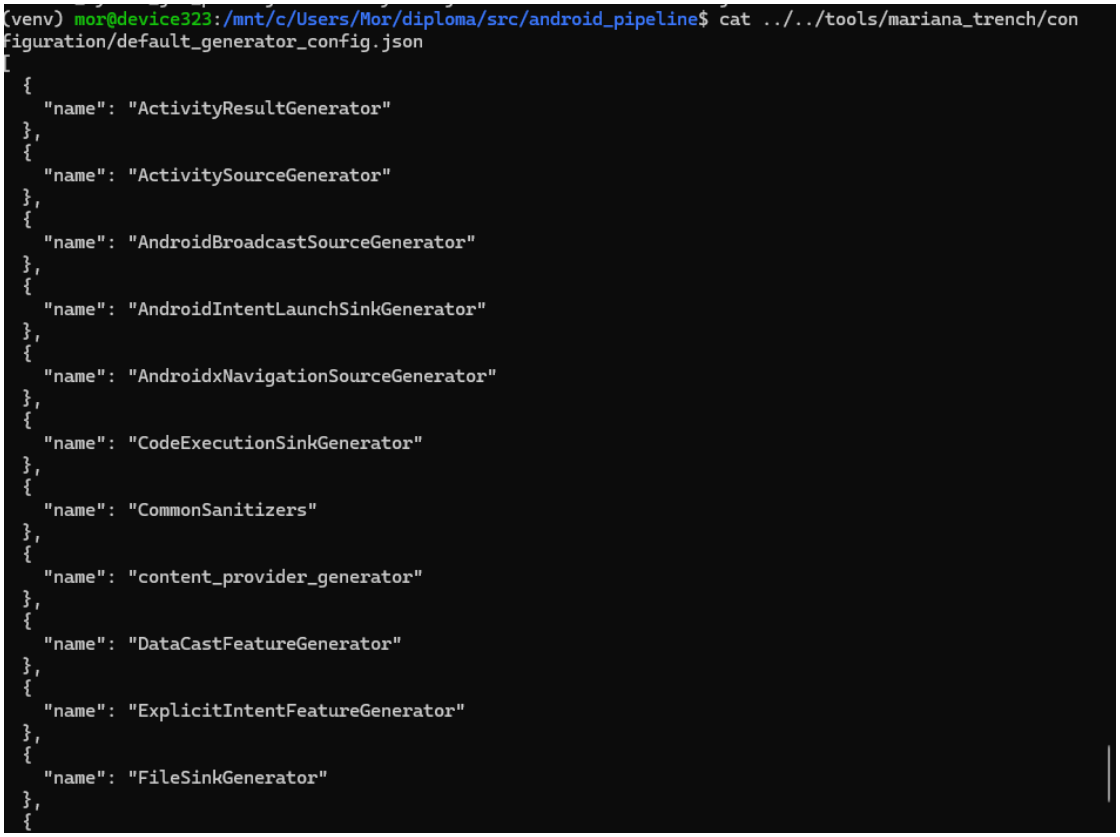


Рисунок 3.3.3 – Фрагмент файлу конфігурації default_generator_config.json

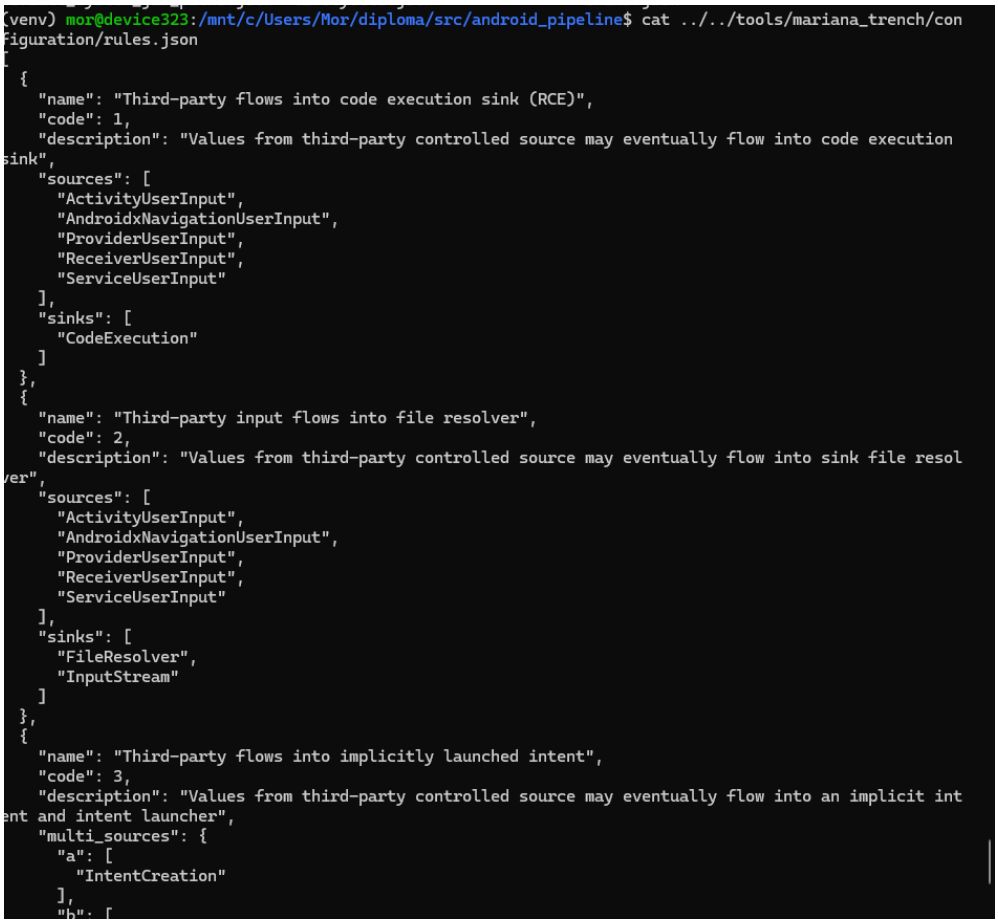


Рисунок 3.3.4 – Фрагмент файлу конфігурації rules.json

Але вихідні дані mariana trench це величезні масиви сирих текстових даних про знайдені вразливості. Читати їх вручну важко. Ось тут врешті решт запускається sapp. Він переганяє результат mariana trench у базу даних(за замовченням у sqlite) для більш швидкого доступу.

```python
def run_mariana_trench(apk_path, out_dir, source_root_dir, work_dir):
    configuration = f"{work_dir}/tools/mariana_trench/configuration"
    system_jar = f"{configuration}/default_system_jar_paths.json"
    model_generator = f"{configuration}/default_generator_config.json"
    rules = f"{configuration}/rules.json"
    lifecycle_paths = f"{configuration}/lifecycles.json"
    model_generator_search = f"{configuration}/model-generators/"

    cmd = (
        f"mariana-trench "
        f"--apk-path {apk_path} "
        f"--output-directory {out_dir} "
        f"--system-jar-configuration-path {system_jar} "
        f"--model-generator-configuration-paths {model_generator} "
        f"--rules-paths {rules} "
        f"--lifecycles-paths {lifecycle_paths} "
        f"--model-generator-search-paths {model_generator_search} "
        f"--source-root-directory {source_root_dir}/sources"
    )
    subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
    print("[*] Mariana Trench finished. Parsing results into SAPP database...")
    sapp_analyze_cmd = f"sapp --tool mariana-trench --database-name {out_dir}/m_trench.db analyze {out
_dir}"
    return subprocess.run(sapp_analyze_cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE
, text=True)
```

Рисунок 3.3.5 – Функція запуску mariana trench та sapp

```
==========================================
[+] STARTING STAGE 2: PARALLEL STATIC ANALYSIS
==========================================
[*] Starting Regex Pattern Matching...
[*] Starting Semgrep AST Analysis...
[+] Regex scan completed. Found 0 potential signature matches.
[+] Semgrep AST Analysis completed.
[+] Correlating Semgrep findings with Dex Call Graph...
```

Рисунок 3.3.6 – Демонстрація роботи другого етапу

Для того щоб візуалізувати таку базу даних потрібно просто запустити веб-інтерфейс sapp командою наступним чином:

```
(venv) mor@device323:/mnt/c/Users/Mor/diploma/src/android_pipeline$ sapp --database-name /mnt/c/Users/
Mor/diploma/output_report/mariana_out/m_trench.db server --source-directory=/mnt/c/Users/Mor/diploma/o
utput_report/jadax_out/sources/
 * Serving Flask app 'sapp.ui.server'
 * Debug mode: off
2026-06-23 21:07:24,944 [INFO] WARNING: This is a development server. Do not use it in a production de
ployment. Use a production WSGI server instead.
 * Running on http://localhost:13337
2026-06-23 21:07:24,944 [INFO] Press CTRL+C to quit
```

Рисунок 3.3.7 – Запуск sapp веб-інтерфейсу

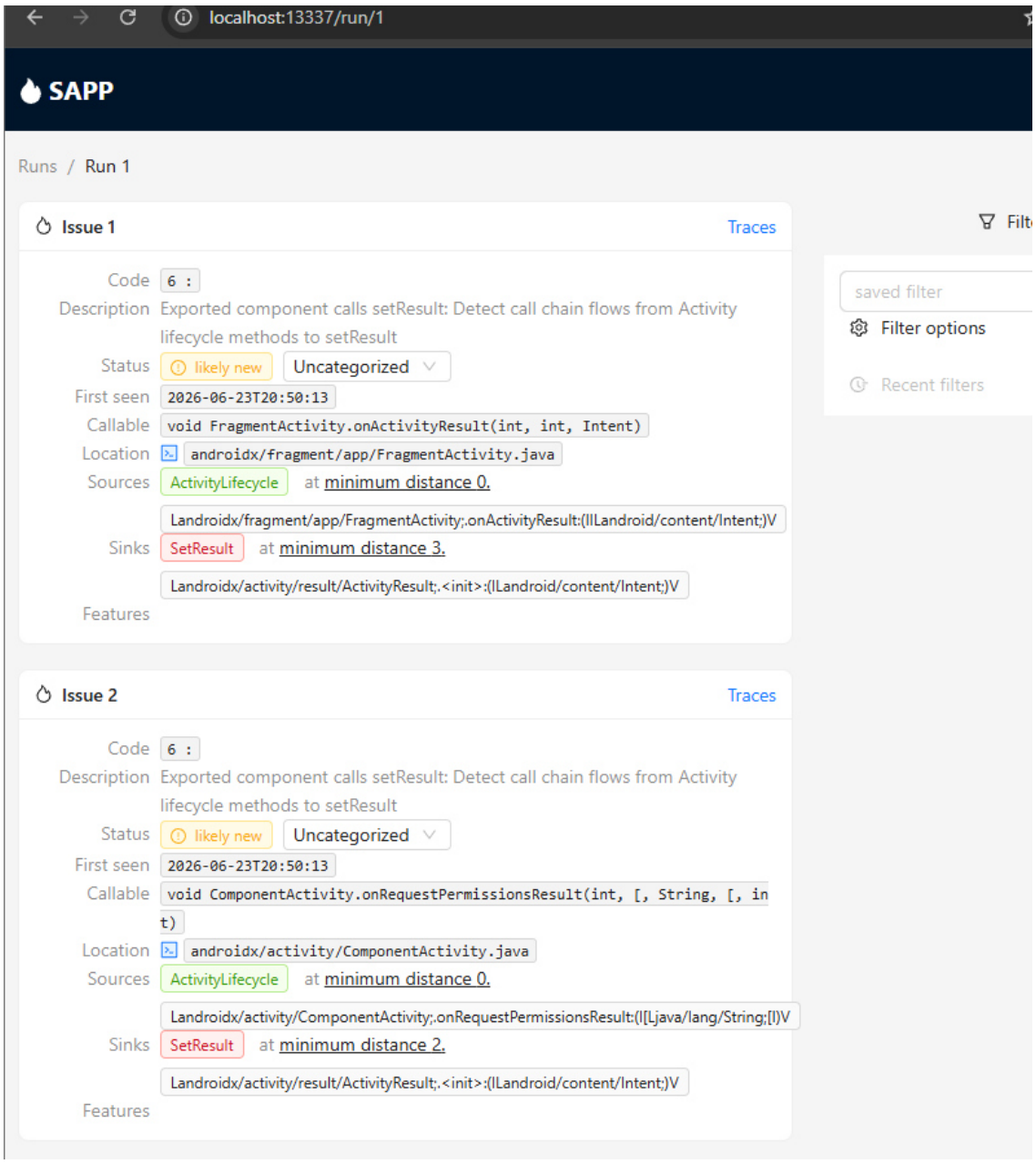


Рисунок 3.3.8 – Перегляд вразливостей у веб-інтерфейсі sapp

3.4 Сигнатурний та семантичний аналіз

Після аналіза потоків даних було реалізовано сигнатурний та семантичний аналіз APK.

Сигнатурний аналіз реалізовано за допомогою регулярних виразів під кожну тему: Мережі, ключи доступу, витоки, слабка криптографія.



```
def run_regex(sources_dir, out_file):
    REGEX_RULES = {
    # --- CREDENTIALS, SECRETS & TOKENS ---
    "AWS_API_Key": r"AKIA[0-9A-Z]{16}",
    "AWS_Secret_Key": r"[0-9a-zA-Z+/]{40}",
    "Google_API_Key": r"AIza[0-9A-Za-z\\-_]{35}",
    "Firebase_URL": r"https://[a-zA-Z0-9\\-_]+\\.firebaseio\\.com",
    "GitHub_PAT": r"ghp_[a-zA-Z0-9]{36}",
    "Slack_Webhook": r"https://hooks\\.slack\\.com/services/T[a-zA-Z0-9_]+/B[a-zA-Z0-9_]+/[a-zA-Z0-
+",
    "JWT_Token": r"ey[A-Za-z0-9\-_=]+\.ey[A-Za-z0-9\-_=]+\.[A-Za-z0-9\-_=]+",
    "Generic_Secret_or_Password": r"(?i)(password|secret|passwd|token|private_key|auth_token|api_se
t)\s*=\s*['\"][A-Za-z0-9+/=_\-]{8,}['\"]",
    "RSA_Private_Key_Header": r"-----BEGIN[^-\r\n]+PRIVATE KEY-----",

    # --- NETWORK & INSECURE INFRASTRUCTURE ---
    "Insecure_HTTP_Protocol": r"\bhttp://[a-zA-Z0-9\\-.&_\\/=?:]+",
    "Internal_IP_Target": r"\b(192\.168\.\d{1,3}\.\d{1,3}|10\.\d{1,3}\.\d{1,3}\.\d{1,3}|172\.(1[6-9
\d|3[01])\.\d{1,3}\.\d{1,3})\b",
    "Bind_All_Interfaces": r"\b0\.0\.0\.0\b",

    # --- WEAK CRYPTOGRAPHY ---
    "Weak_Hash_MD5": r"(?i)MessageDigest\.getInstance\(\s*['\"]MD5['\"]",
    "Weak_Hash_SHA1": r"(?i)MessageDigest\.getInstance\(\s*['\"]SHA-1['\"]",
    "Insecure_Cipher_AES_ECB": r"(?i)AES/ECB/(PKCS5Padding|NoPadding)",

    # --- INFORMATION LEAKAGE & DIAGNOSTICS ---
    "Android_Log_Leak": r"\bLog\.(e|d|v|i|wtf)\s*\(",
    "Java_PrintStackTrace": r"\.printStackTrace\s*\(\s*\)",
    "System_Out_Err_Print": r"\bSystem\.(out|err)\.print",

    # --- ANDROID MANIFEST CONFIGURATION FLAWS ---
    "Manifest_AllowBackup_Enabled": r"android:allowBackup\s*=\s*['\"]true['\"]",
    "Manifest_Debuggable_Enabled": r"android:debuggable\s*=\s*['\"]true['\"]",
    "Manifest_CleartextTraffic_Allowed": r"android:usesCleartextTraffic\s*=\s*['\"]true['\"]",
    "Manifest_ExportedComponent_Unprotected": r"android:exported\s*=\s*['\"]true['\"]"
}
    print("[*] Starting Regex Pattern Matching...")
    findings = []

    for root, _, files in os.walk(sources_dir):
        for file in files:
            if file.endswith(".java") or file == "AndroidManifest.xml":
                file_path = os.path.join(root, file)
                try:
                    with open(file_path, "r", encoding="utf-8", errors="ignore") as f:
                        for line_num, line in enumerate(f, 1):
                            clean_line = line.strip()
                            for rule_name, pattern in REGEX_RULES.items():
                                                                    24,18         1
```

Рисунок 3.4.1– Реалізація сигнатурного аналізу

```
                with open(file_path, "r", encoding="utf-8", errors="ignore") as f:
                    for line_num, line in enumerate(f, 1):
                        clean_line = line.strip()
                        for rule_name, pattern in REGEX_RULES.items():
                            matches = re.finditer(pattern, clean_line)
                            for match in matches:
                                findings.append({
                                    "rule": rule_name,
                                    "file": os.path.relpath(file_path, sources_dir),
                                    "line": line_num,
                                    "match": match.group(0)[:150]
                                })
            except Exception as e:
                pass

    with open(out_file, "w", encoding="utf-8") as f:
        json.dump(findings, f, indent=4, ensure_ascii=False)

    print(f"[+] Regex scan completed. Found {len(findings)} potential signature matches.")
```

Рисунок 3.4.2 – Реалізація сигнатурного аналізу

Для сематичного аналізу було використано інструмент semgrep.

```python
def run_semgrep(sources_dir, out_file, work_dir):
    print("[*] Starting Semgrep AST Analysis...")
    cmd = f"semgrep scan -c {work_dir}/tools/rules/ --json -o {out_file} {sources_dir}"

    result = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True
)

    if result.returncode not in [0, 1]:
        print(f"[-] Semgrep failed: {result.stderr}")
    else:
        print("[+] Semgrep AST Analysis completed.")
```

Рисунок 3.4.3 – Функція семантичного аналізу за допомогою semgrep

Semgrep простий у використанні але потребує багато правил. У рамках даної роботи було створено правила як для xml файлів так і для java.

```yaml
rules:
  - id: insecure_cipher_3des
    severity: WARNING
    languages:
      - java
      - kotlin
    metadata:
      category: security
      confidence: HIGH
      likelihood: HIGH
      impact: HIGH
      subcategory:
        - vuln
      owasp-mas:
        identifier: MASVS-CRYPTO-1
        area: cryptography
        level:
          - L1
          - L2
      references:
        - https://mas.owasp.org/MASTG/General/0x04g-Testing-Cryptography/#identifying-insecure-andor-d
eprecated-cryptographic-algorithms
        - https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html#cipher
-modes
    message: >-
      This Android mobile application seems using a weak cipher, `3DES`, which could expose to various
 kind of vulnerabilities.
      The data encrypted using `3DES` mode block could allow to retrieve its plain-text value.
      It is recommended to avoid the use of `3DES`, and use `AES` instead.
    pattern-either:
        - pattern: Cipher $VAR = Cipher.getInstance("=~/((3DES|DESede|DESEDEWRAP)\/.*\/.*|(3DES|DESede
|DESEDEWRAP))/i", ...);
        - pattern: SecretKeyFactory $VAR = SecretKeyFactory.getInstance("=~/(DESede\/.*\/.*|DESede)$/i
", ...);
        - pattern: $VAR = Cipher.getInstance("=~/((3DES|DESede|DESEDEWRAP)\/.*\/.*|(3DES|DESede|DESEDE
WRAP))/i", ...);
        - pattern: $VAR = SecretKeyFactory.getInstance("=~/(DESede\/.*\/.*|DESede)$/i", ...);
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

Рисунок 3.4.4 – Правило для java-файлів

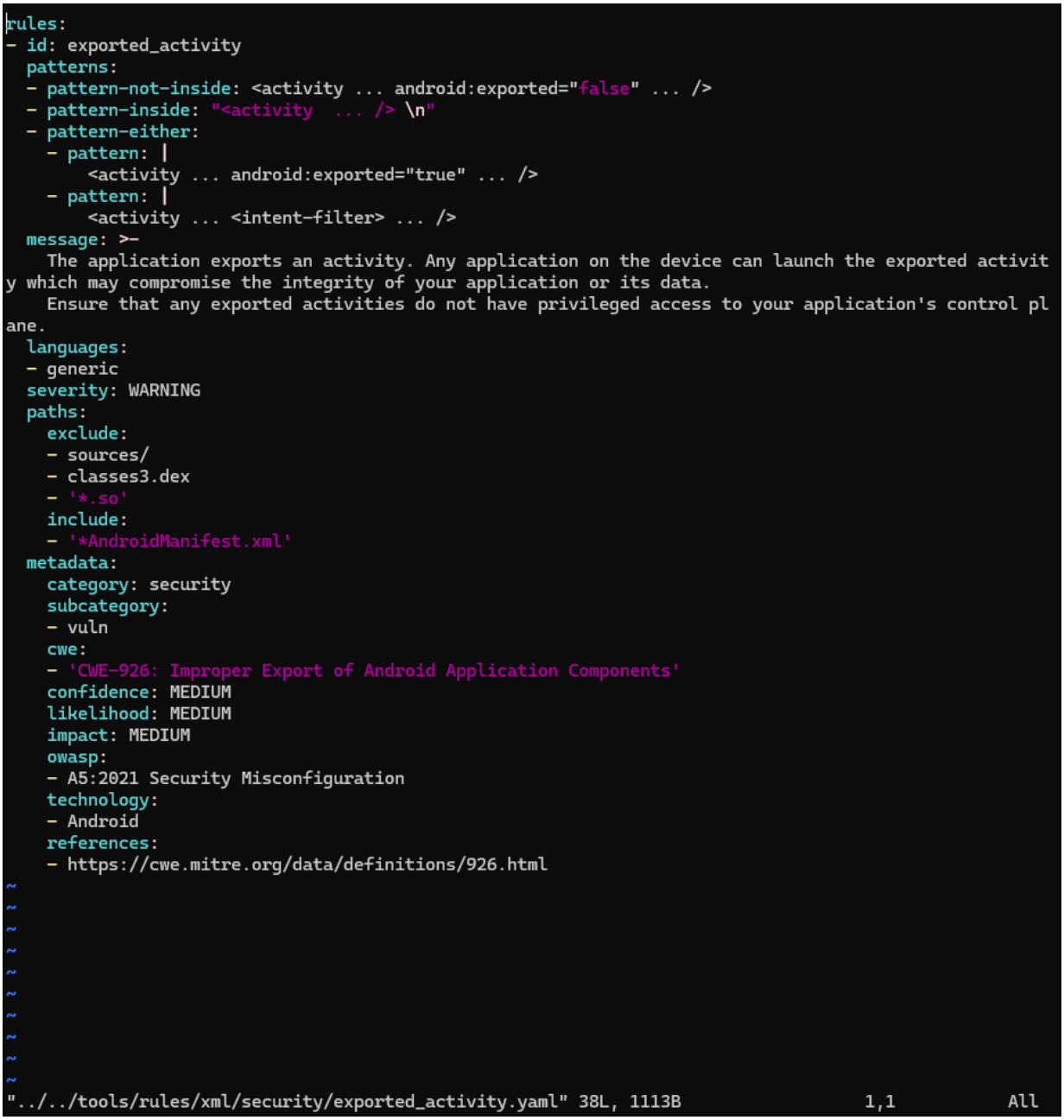


Рисунок 3.4.5 – Правило для xml-файлів

Після семантичного аналізу запускається модуль спеціалізованого модуля аналізу досяжності. Ця логіка повноцінно зорієнтована на глибоке трасування методів та побудову ланцюжків викликів за допомогою функціональних можливостей підсистеми `AnalyzeAPK` аналітичної бібліотеки `androguard`. Програмна логіка третього етапу інтегрована у функцію `run_reachability_analysis` та виконується за таким алгоритмом:

1. Ініціалізація низькорівневого графа викликів: За допомогою методу `AnalyzeAPK` інструментарій `androguard` здійснює повний розбір байт-коду

APK-файла, формуючи об'єкти аналізу додатку та об'єкт аналізу внутрішніх зв'язків і крос-посилань.

2. Фільтрація та детермінація цільових класів: Програма ітерує кожну виявлену проблему, вилучаючи шлях до файлу, номер рядка коду та унікальний ідентифікатор правила безпеки. Для маппінгу вихідного коду на бінарні структури всередині DEX-файла, скрипт динамічно виокремлює назву класу та трансформує її у внутрішній специфічний формат іменування класів віртуальної машини Dalvik.

3. Пошук точок входу та відсікання мертвого коду: Визначена назва цільового класу передається до внутрішнього реєстру об'єкта аналізу. Якщо підсистема `androguard` не знаходить відповідний клас у згенерованому DEX-графі, скрипт ідентифікує цей сегмент як «мертвий код», що дозволяє суттєво знизити кількість хибнопозитивних спрацювань.

```
import json
import os
from pathlib import Path
from androguard.misc import AnalyzeAPK

def run_reachability_analysis(apk_path, semgrep_report_path, output_report_path):
    print("\n" + "="*50)
    print("[+] STARTING STAGE 3: REACHABILITY & CALL CHAIN TRACING")
    print("="*50)

    if not os.path.exists(semgrep_report_path):
        print(f"[-] Semgrep report not found at {semgrep_report_path}. Skipping Stage 3.")
        return

    with open(semgrep_report_path, "r", encoding="utf-8") as f:
        semgrep_data = json.load(f)

    findings = semgrep_data.get("results", [])
    if not findings:
        print("[+] Semgrep found 0 issues. Nothing to trace.")
        return

    print(f"[*] Loaded {len(findings)} Semgrep findings. Initializing Androguard Call Graph...")

    try:
        a, d, dx = AnalyzeAPK(apk_path)
    except Exception as e:
        print(f"[-] Androguard failed to analyze APK: {e}")
        return

    trace_results = []

    for issue in findings:
        file_path = issue.get("path", "")
        line_num = issue.get("start", {}).get("line", 0)
        rule_id = issue.get("check_id", "Unknown_Rule")

        if "sources/" in file_path:
            class_part = file_path.split("sources/")[-1].replace(".java", "")
            target_class_name = f"L{class_part};"
        else:
            continue

        print(f"[*] Tracing call chains for {rule_id} found in {target_class_name} at line {line_num}
)

        analyzed_class = dx.classes.get(target_class_name)
        if not analyzed_class:
            print(f"    [-] Class {target_class_name} not found in DEX graph (could be dead code).")
            continue
```

Рисунок 3.4.6 – Реалізація трасування вразливих методів

```
        file_path = issue.get("path", "")
        line_num = issue.get("start", {}).get("line", 0)
        rule_id = issue.get("check_id", "Unknown_Rule")

        if "sources/" in file_path:
            class_part = file_path.split("sources/")[-1].replace(".java", "")
            target_class_name = f"L{class_part};"
        else:
            continue

        print(f"[*] Tracing call chains for {rule_id} found in {target_class_name} at line {line_num}"
)


        analyzed_class = dx.classes.get(target_class_name)
        if not analyzed_class:
            print(f"    [-] Class {target_class_name} not found in DEX graph (could be dead code).")
            continue

        class_chains = []
        for method in analyzed_class.get_methods():
            xrefs_from = method.get_xref_from()

            for ref_class, ref_method, offset in xrefs_from:
                chain_entry = {
                    "called_by_class": ref_class.name,
                    "called_by_method": ref_method.name,
                    "via_method": method.name
                }
                class_chains.append(chain_entry)
                print(f"    [!] Trigger Path Found: {ref_class.name}->{ref_method.name}() calls {metho
d.name}()")

        trace_results.append({
            "rule_id": rule_id,
            "target_class": target_class_name,
            "line": line_num,
            "is_reachable": len(class_chains) > 0,
            "severity": "CRITICAL" if len(class_chains) > 0 else "WARNING (Unreachable/Shielded)",
            "call_connections": class_chains
        })

    with open(output_report_path, "w", encoding="utf-8") as f:
        json.dump(trace_results, f, indent=4, ensure_ascii=False)

    print(f"\n[+] Reachability analysis completed. Report saved to: {output_report_path}")
~
```

Рисунок 3.4.7 – Реалізація трасування вразливих методів

3.5 Перевірка результатів аналізу

Як було детально обґрунтовано та покроково розписано у попередньому пункті 3.2, усі підсумкові звіти, проміжні конфігураційні файли та логи, що формуються в процесі послідовного або паралельного виконання етапів пайплайну, консолідуються та автоматично зберігаються в межах цільової директорії, яка вказується як обов'язковий аргумент безпосередньо під час ініціалізації та старту системи. Результати роботи sapp, mariana trench та парсеру метаданих було продемонстровано у пунктах 3.3 та 3.2 відповідно.

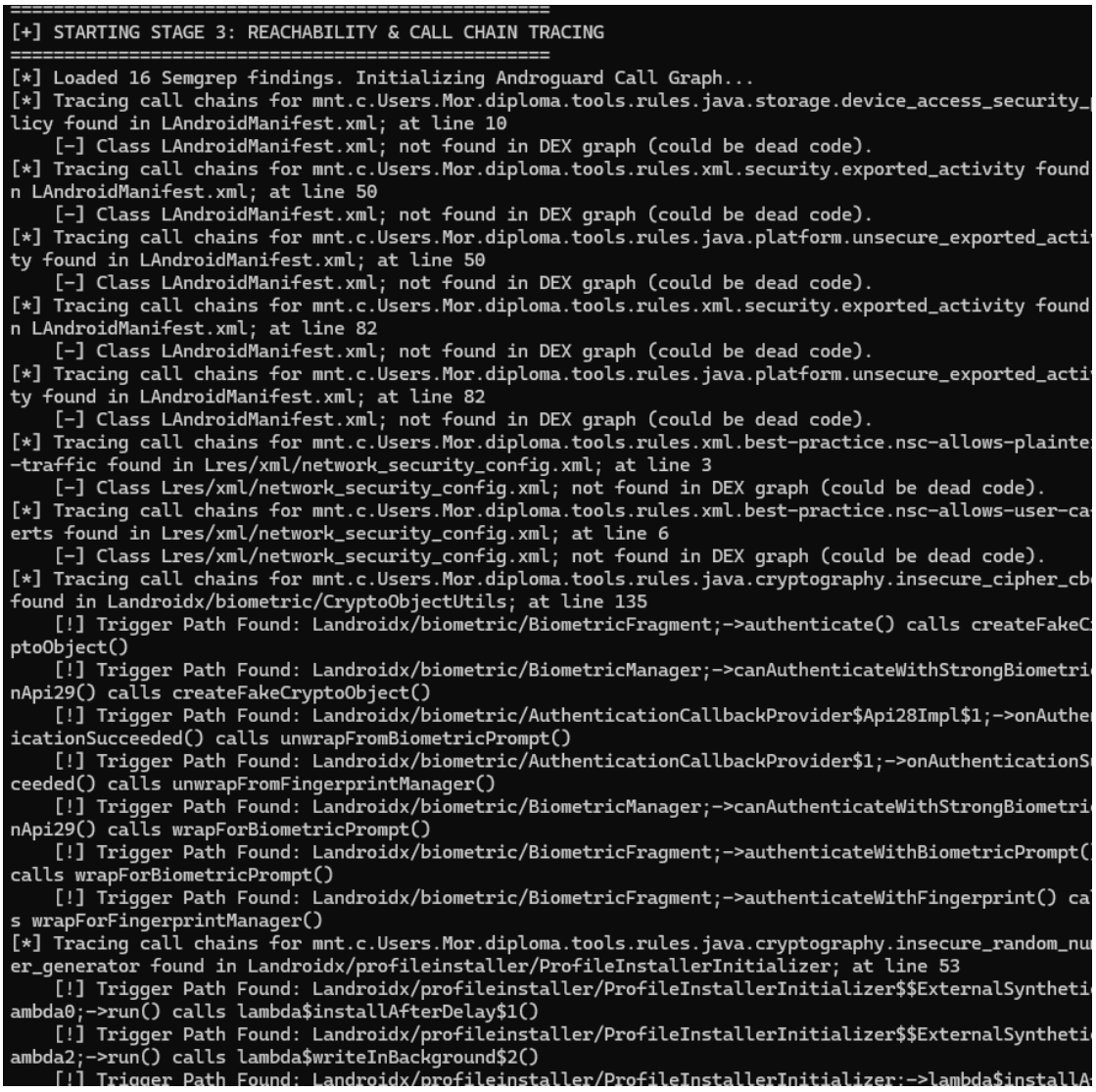


Рисунок 3.5.1 – Відпрацювання semgrep та трасування методів

Як можна побачити деякі вразливості не відтрасувались але це не є помилкою. Не трасуються xml-файли адже вони є файлами розмітки/конфігурації.

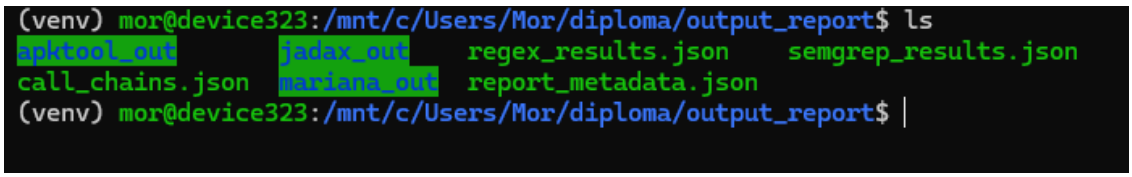


Рисунок 3.5.2 – Перегляд вмісту вихідного каталогу

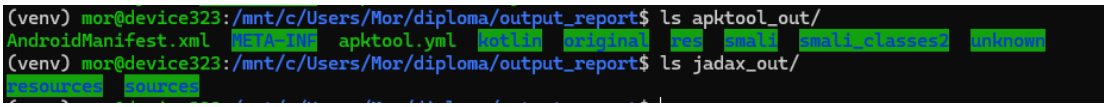


Рисунок 3.5.3 – Перегляд вмісту вихідних каталогів декомпіляторів

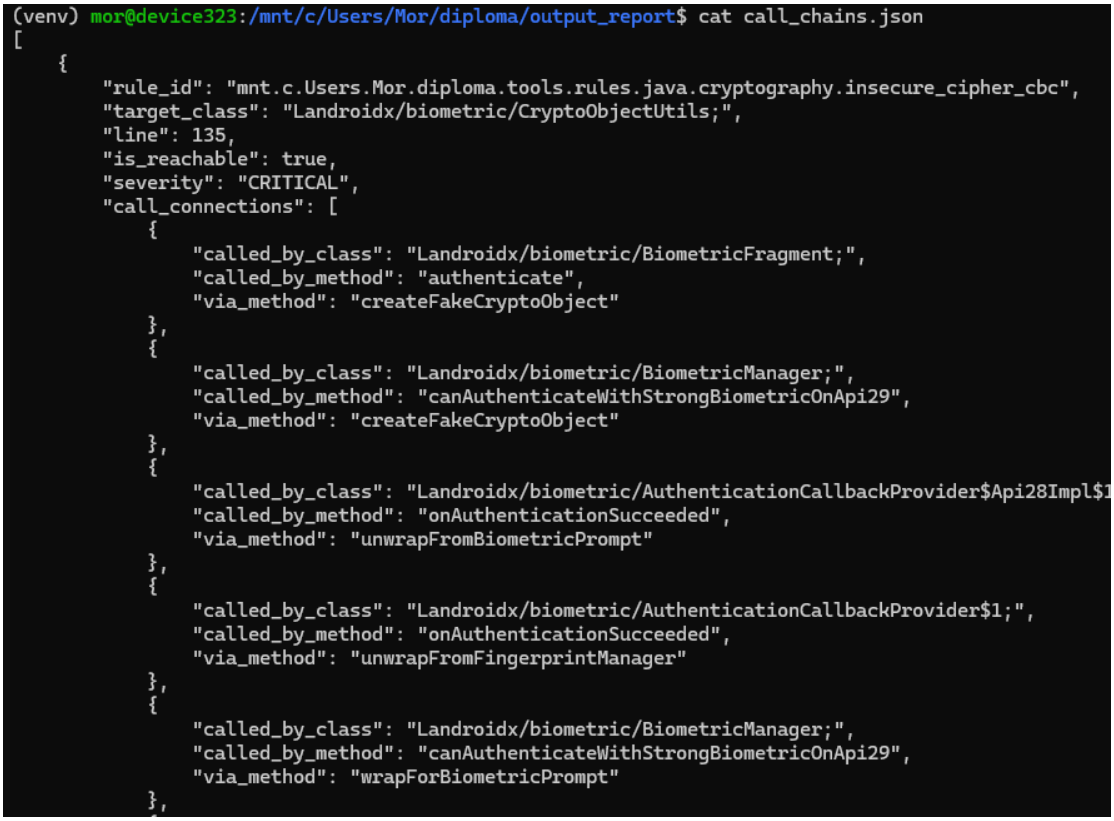


Рисунок 3.5.4 – Перегляд вмісту вихідного файлу трасування

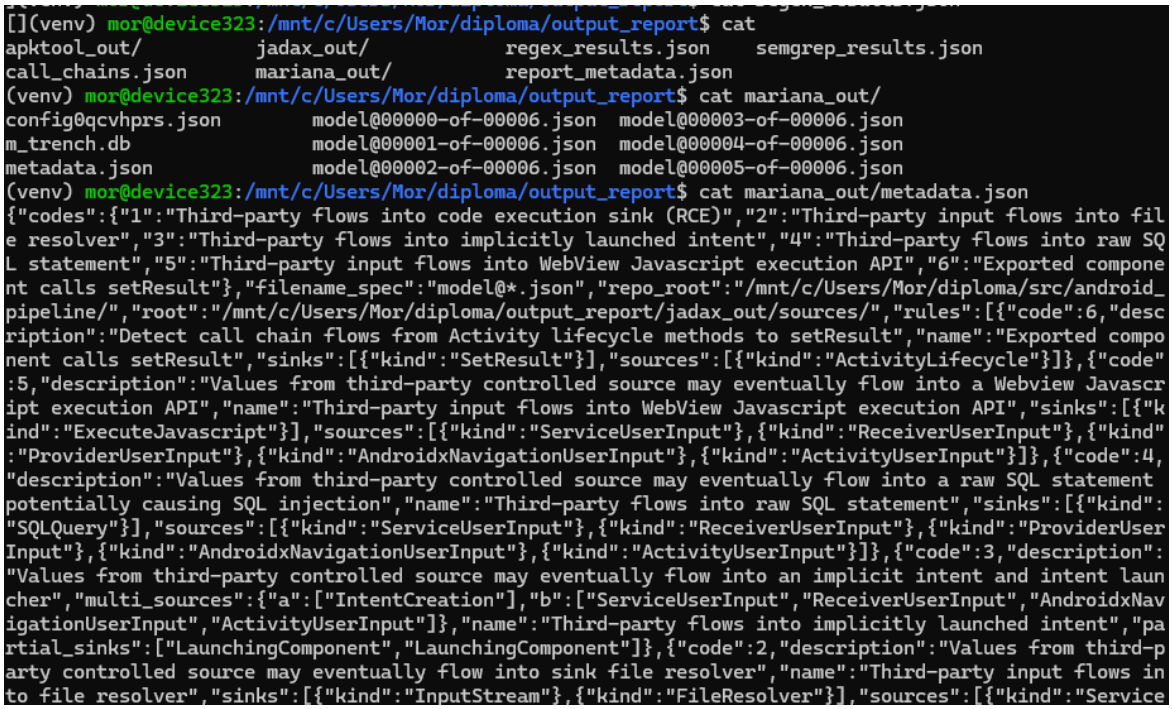


Рисунок 3.5.5 – Перегляд вмісту вихідного файлу аналізу передачі даних



Рисунок 3.5.6 Перегляд вмісту вихідного файлу semgrep

Представлені вище лістинги лог-файлів, структури директорій та вихідні артефакти є прямим інструментальним підтвердженням повної працездатності, стабільності та архітектурної цілісності розробленого автоматизованого пайплайну аналізу захищеності. Проведений моніторинг журналу подій та верифікація вихідних даних дозволяють констатувати, що кожен із інтегрованих модулів – включаючи статичний аналізатор Mariana Trench, систему оркестрації та менеджменту результатів SAPP, а також кастомний парсер метаданих – успішно завершив свій життєвий цикл виконання без виникнення критичних збоїв, переривань або аномальних відхилень від заданого алгоритму.

Усі згенеровані файли логів та структуровані звіти у цільовій директорії повністю відповідають специфікаціям вхідних параметрів, заданих при ініціалізації процесу.

Таким чином, успішне логування та фіксація результатів роботи утиліт підтверджує, що розроблений комплекс інструментів готовий до подальшої практичної експлуатації.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи досягнута поставлена мета, що включала розробку, інтеграцію та практичне застосування автоматизованого комплексу аналізу захищеності мобільних додатків з метою виявлення конфігураційних ризиків та архітектурних уразливостей логічної структури коду. У роботі детально розглянуто фундаментальні поняття безпеки мобільних екосистем, методи дослідження компілятів прикладного програмного забезпечення, а також детермінацію виникнення дефектів внаслідок впливу антропогенного фактора на етапах проектування та кодування.

У ході роботи було проаналізовано архітектуру сучасного мобільного програмного забезпечення, що дозволило визначити логічні зв’язки між окремими компонентами систем. На основі проведеного аналізу сформовано послідовність дій та інструментальний стек, необхідні для ефективного статичного дослідження мобільних додатків за допомогою методів SAST та аналізу руху даних, що дозволяє виявляти критичні вразливості та системні прогалини до етапу розгортання продукту на пристроях кінцевих користувачів.

Після вивчення теоретичних підходів було здійснено практичну реалізацію та розгортання автоматизованого пайплайну аналізу, який включав інтеграцію статичного аналізатора глибокого аналізу потоків даних Mariana Trench, системи оркестрації та менеджменту результатів SAPP, а також спеціалізованого кастомного парсеру метаданих і дескрипторів розгортання. У процесі дослідження було виконано детекцію критичних конфігураційних помилок, відтворено повний ланцюг обробки результатів сканування, а також проведено комплексну верифікацію та тріаж виявлених дефектів. Отримані результати дозволили автоматизувати процес ідентифікації архітектурних недоліків та структурувати лог-файли у цільових директоріях.

Практична частина роботи продемонструвала, що складні архітектурні дефекти та конфігураційні ризики мобільних додатків можуть бути ефективно ідентифіковані за допомогою розробленого автоматизованого комплексу.

Отримані результати та розроблена методологія побудови пайплайну мають прямі застосування у практичній діяльності компаній з аудиту кібербезпеки, підготовці спеціалістів у сфері безпечної розробки, а також можуть бути адаптовані для розширення інструментарію автоматизованого контролю якості коду. Перспективними напрямами розвитку є автоматизація процесів написання кастомних правил аналізу, розширення інструментарію для аналізу кросплатформених мобільних фреймворків, а також інтеграція розробленого пайплайну у сучасні хмарні системи безперервної інтеграції та доставки.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Android Application Sandbox: Architecture, Isolation, and UID Management. *Android Open Source Project (AOSP).* URL: https://source.android.com/docs/security/app-sandbox
2. Permissions on Android: Security Model, User Authorization, and Protection Levels. *Android Developers Guide.* . URL: https://developer.android.com/guide/topics/security/permissions
3. Security-Enhanced Linux in Android: Concepts, Implementation, and Policies. *Android Open Source Project (AOSP).* URL: https://source.android.com/docs/security/features/selinux
4. Protection Levels and Signature Permissions in Android. *Android Developers Guide*. URL: https://developer.android.com/guide/topics/manifest/permission-element
5. APK Signature Scheme v3: Support for Certificate Rotation and Signing History. *Android Open Source Project (AOSP).* URL: https://source.android.com/docs/security/features/apksigning/v3
6. Structure of an Android App Binary (.apk): Structural Components and Internal Anatomy. *Appdome DevSecOps Knowledge Base*. URL: https://www.appdome.com/how-to/devsecops-automation-mobile-cicd/appdome-basics/structure-of-an-android-app-binary-apk/
7. Dissecting an APK: A Malware Analyst's Perspective. *The Malware Files.* URL: https://themalwarefiles.com/dissecting-an-apk-a-malware-analysts-perspective-ab33b48d2333
8. Android Tampering and Reverse Engineering. *Mobile Security Testing Guide*. URL: https://pentest.y-security.de/Mobile%20Security%20Testing%20Guide/2023/0x05c-Reverse-Engineering-and-Tampering/
9. Exploiting the Native Libraries of an Android app. *HASHNODEDEV*. URL: https://dispatchersdotplayground.hashnode.dev/exploiting-the-native-libraries-of-an-android-app

10. The Role of Human Factors in Cybersecurity Vulnerabilities. *ResearchGate*. URL: https://www.researchgate.net/publication/384621226_The_Role_of_Human_Factors_in_Cybersecurity_Vulnerabilities

11. Android:Debuggable | *Security. Android Developers*. URL: https://developer.android.com/privacy-and-security/risks/android-debuggable

12. MASTG-KNOW-0014: Android Network Security Configuration. *OWASP MAS*. URL: https://mas.owasp.org/MASTG/knowledge/android/MASVS-NETWORK/MASTG-KNOW-0014/

13. The Hidden Threat of Overprivileged Apps. *Quokka.io*. URL: https://www.quokka.io/blog/overprivileged-mobile-apps

14. Exported Components Vulnerability in Android. *SecureFlag Knowledge Base*. URL: https://knowledge-base.secureflag.com/vulnerabilities/broken_authorization/exported_components_in_android_vulnerability.html

15. Janus Vulnerability: Threats and Mitigation. *GuardSquare*. URL: https://www.guardsquare.com/blog/new-android-vulnerability-allows-attackers-to-modify-apps-without-affecting-their-signatures-guardsquare

16. Protect and Safeguard API Keys from Leakage and Misuse. *GuardSquare*. URL: https://www.guardsquare.com/blog/protect-api-keys-from-leaks

17. OWASP Mobile Application Security Testing Guide. *OWASP Foundation*. URL: https://mas.owasp.org/MASTG/

18. OWASP Mobile Application Security Verification Standard . *OWASP Foundation*. URL: https://mas.owasp.org/MASVS/

19. Vetting the Security of Mobile Applications, Publication 800-163 Revision 1 . *NIST*. URL: https://csrc.nist.gov/pubs/sp/800/163/r1/final

20. Protection Profile for Application Software. *CommonCriteriaPortal*. URL: https://www.commoncriteriaportal.org/files/ppfiles/pp_app_v1.4.pdf

21. PCI Mobile Payment Acceptance Security Guidelines for Developers. *PCI Security Standards Council*. URL: https://www.pcisecuritystandards.org/documents/Mobile_Payment_Acceptance_Security_Guidelines.pdf
22. How to Use JADX – Dex to Java Decompiler. *Baeldung*. URL: https://www.baeldung.com/java-jdx-decompile-classes
23. Decompile and Recompile APK using APKTOOL – Beginner's Guide. *Medium*. URL: https://medium.com/@ps.sujith/decompile-and-recompile-apk-using-apktool-beginners-guide-4ad03c2c5b8f
24. Smith W. Semgrep in Practice: The Complete Guide for Developers and Engineers
25. Static Analysis With Mobile Security Framework (MobSF) Part-1. *Medium*. URL: https://medium.com/@abwahab5095/getting-started-with-mobile-security-framework-mobsf-part-1-7fc422d44164
26. Dynamic Instrumentation on Android with Frida: A Practical Guide. *Medium*. URL: https://can-ozkan.medium.com/dynamic-instrumentation-on-android-with-frida-a-practical-guide-878f492144ff
27. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Applications. *ACM DL*. URL: https://dl.acm.org/doi/10.1145/2666356.2594299
28. Mobile App Pentesting: AndroGOAT Assessment Walkthrough. *InfoSec Write-ups*. URL: https://infosecwriteups.com/mobile-pentesting-androgoat-assessment-walkthrough-1a63a7edc677
29. Android Penetration Tools Walkthrough Series: Androguard. *InfoSec Institute*. URL: https://www.infosecinstitute.com/resources/penetration-testing/android-penetration-tools-walkthrough-series-androguard/

ДОДАТКИ

Додаток А

Скрипт dex_tracer.py

```
import json
import os
from pathlib import Path
from androguard.misc import AnalyzeAPK

def run_reachability_analysis(apk_path, semgrep_report_path, output_report_path):
    print("\n" + "="*50)
    print("[+] STARTING STAGE 3: REACHABILITY & CALL CHAIN TRACING")
    print("="*50)

    if not os.path.exists(semgrep_report_path):
        print(f"[-] Semgrep report not found at {semgrep_report_path}. Skipping Stage 3.")
        return

    with open(semgrep_report_path, "r", encoding="utf-8") as f:
        semgrep_data = json.load(f)

    findings = semgrep_data.get("results", [])
    if not findings:
        print("[+] Semgrep found 0 issues. Nothing to trace.")
        return

    print(f"[*] Loaded {len(findings)} Semgrep findings. Initializing Androguard Call Graph...")

    try:
        a, d, dx = AnalyzeAPK(apk_path)
    except Exception as e:
        print(f"[-] Androguard failed to analyze APK: {e}")
        return

    trace_results = []

    for issue in findings:
        file_path = issue.get("path", "")
        line_num = issue.get("start", {}).get("line", 0)
        rule_id = issue.get("check_id", "Unknown_Rule")

        if "sources/" in file_path:
            class_part = file_path.split("sources/")[-1].replace(".java", "")
            target_class_name = f"L{class_part};"
        else:
            continue

        print(f"[*] Tracing call chains for {rule_id} found in {target_class_name} at line {line_num}")

        analyzed_class = dx.classes.get(target_class_name)
        if not analyzed_class:
            print(f"    [-] Class {target_class_name} not found in DEX graph (could be dead code).")
            continue
```

```
        class_chains = []
        for method in analyzed_class.get_methods():
            xrefs_from = method.get_xref_from()

            for ref_class, ref_method, offset in xrefs_from:
                chain_entry = {
                    "called_by_class": ref_class.name,
                    "called_by_method": ref_method.name,
                    "via_method": method.name
                }
                class_chains.append(chain_entry)
                  print(f"    [!] Trigger Path Found: {ref_class.name}->{ref_method.name}() calls {method.-
name}()")

        trace_results.append({
            "rule_id": rule_id,
            "target_class": target_class_name,
            "line": line_num,
            "is_reachable": len(class_chains) > 0,
            "severity": "CRITICAL" if len(class_chains) > 0 else "WARNING (Unreachable/Shielded)",
            "call_connections": class_chains
        })

    with open(output_report_path, "w", encoding="utf-8") as f:
        json.dump(trace_results, f, indent=4, ensure_ascii=False)

print(f"\n[+] Reachability analysis completed. Report saved to: {output_report_path}")
```

Скрипт main.py

```
import argparse
import sys
import time
from pathlib import Path

from utils import reset_directory, save_metadata
from stage0 import run_stage_0
from stage1 import run_stage_1
from stage2 import run_stage_2


def main():
    parser = argparse.ArgumentParser(
        description="Android Automated Static Analysis Pipeline (Stage 0 + Stage 1)"
    )
    parser.add_argument(
        "--apk",
        required=True,
        help="Path to the target APK file"
    )
    parser.add_argument(
        "--workdir",
        required=True,
        help="Absolute path to the working directory"
```

```
)

args = parser.parse_args()

apk_path = Path(args.apk)
if not apk_path.exists():
    print(f"[-] Error: APK file not found at {args.apk}")
    sys.exit(1)

work_dir = Path(args.workdir)
out_dir = work_dir / "output_report"

print("=" * 60)
print("=== STARTING AUTOMATED APK ANALYSIS PIPELINE ===")
print("=" * 60)

total_start_time = time.time()
reset_directory(out_dir)
# ------------------------------------------------------------
# STAGE 0:
# ------------------------------------------------------------
print("[*] Launching Stage 0: Extracting APK Metadata...")
try:
    metadata = run_stage_0(str(apk_path))
    out_dir.mkdir(parents=True, exist_ok=True)
    save_metadata(metadata, out_dir)
except Exception as e:
    print(f"[-] Critical Error in Stage 0: {e}")
    sys.exit(1)

print("-" * 60)

# ------------------------------------------------------------
# STAGE 1:
# ------------------------------------------------------------
print("[*] Launching Stage 1: Decompilation & Apk Scanning...")
try:
    run_stage_1(str(apk_path), str(work_dir), Path(out_dir))
except Exception as e:
    print(f"[-] Critical Error in Stage 1: {e}")
    sys.exit(1)

print("-" * 60)
# ------------------------------------------------------------
# STAGE 2:
# ------------------------------------------------------------
print("[*] Launching Stage 2: Static Analysis...")
jadx_sources = f"{work_dir}/output_report/jadax_out/"
try:
    run_stage_2(str(apk_path), str(work_dir), Path(out_dir), jadx_sources)
except Exception as e:
    print(f"[-] Critical Error in Stage 2: {e}")
```

```
        sys.exit(1)

    print("=" * 60)
    print(f"[+] Pipeline finished in {time.time() - total_start_time:.2f}s")
    print(f"[+] All reports are located in: {out_dir}")
    print("=" * 60)



if __name__ == "__main__":
main()
```

Скрипт utils.py

```
import shutil
from pathlib import Path
import json

def save_metadata(metadata, out_dir):
    print("\n[+] Stage 0 Metadata Results:")
    print(json.dumps(metadata, indent=4, ensure_ascii=False))

    try:
        report_file = Path(out_dir) / "report_metadata.json"
        with open(report_file, "w", encoding="utf-8") as f:
            json.dump(metadata, f, indent=4, ensure_ascii=False)
        print(f"[+] Metadata successfully saved to: {report_file}")
    except Exception as e:
        print(f"[-] Warning: Could not save metadata file: {e}")


def reset_directory(dir_path: str | Path):
    path = Path(dir_path)
    if path.exists() and path.is_dir():
        shutil.rmtree(path)
path.mkdir(parents=True, exist_ok=True)
```

Скрипт stage0.py

```
from androguard.core.apk import APK
import time

from loguru import logger
logger.disable("androguard")
import logging
logging.getLogger("androguard").setLevel(logging.WARNING)
def run_stage_0(apk_path):
    start_time = time.time()
    apk = APK(apk_path)

    ns_android = "{http://schemas.android.com/apk/res/android}"

    manifest_xml = apk.get_android_manifest_xml()
```

```
    is_debuggable = False
    application_tag = manifest_xml.find("application")
    if application_tag is not None:
        debuggable_attr = application_tag.get(f"{ns_android}debuggable")
        is_debuggable = debuggable_attr == "true"

    metadata = {
        "package": apk.get_package(),
        "min_sdk": apk.get_min_sdk_version(),
        "target_sdk": apk.get_target_sdk_version(),
        "permissions": apk.get_permissions(),
        "is_debuggable": is_debuggable,
        "exported_components": []
    }

    metadata["v1_signature"] = apk.is_signed_v1()
    metadata["v2_signature"] = apk.is_signed_v2()
    metadata["v3_signature"] = apk.is_signed_v3()

    if metadata["v1_signature"] and not (metadata["v2_signature"] or metadata["v3_signature"]):
        metadata["janus_vulnerable"] = True

    print(f"[*] Stage 0 successfully completed {time.time() - start_time:.4f}c")
return metadata
```

Скрипт stage1.py

```
import subprocess
from concurrent.futures import ThreadPoolExecutor
from pathlib import Path


def run_jadx(apk_path, out_dir, work_dir):
    cmd = f"{work_dir}/tools/jadx/bin/jadx -d {out_dir} -j 8 {apk_path}"
    subprocess.run(cmd, shell=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

def run_apktool(apk_path, out_dir):
    cmd = f"apktool d {apk_path} -o {out_dir} -f"
    subprocess.run(cmd, shell=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

def run_mariana_trench(apk_path, out_dir, source_root_dir, work_dir):
    configuration = f"{work_dir}/tools/mariana_trench/configuration"
    system_jar = f"{configuration}/default_system_jar_paths.json"
    model_generator = f"{configuration}/default_generator_config.json"
    rules = f"{configuration}/rules.json"
    lifecycle_paths = f"{configuration}/lifecycles.json"
    model_generator_search = f"{configuration}/model-generators/"

    cmd = (
        f"mariana-trench "
        f"--apk-path {apk_path} "
        f"--output-directory {out_dir} "
```

```
        f"--system-jar-configuration-path {system_jar} "
        f"--model-generator-configuration-paths {model_generator} "
        f"--rules-paths {rules} "
        f"--lifecycles-paths {lifecycle_paths} "
        f"--model-generator-search-paths {model_generator_search} "
        f"--source-root-directory {source_root_dir}/sources"
    )
    subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
    print("[*] Mariana Trench finished. Parsing results into SAPP database...")
     sapp_analyze_cmd = f"sapp --tool mariana-trench --database-name {out_dir}/m_trench.db analyze {out_dir}"
        return subprocess.run(sapp_analyze_cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)


def run_stage_1(apk_path, work_dir, out_dir):
    out_dir_jadax = out_dir / "jadax_out"
    out_dir_apktool = out_dir / "apktool_out"
    out_dir_mariana = out_dir / "mariana_out"

    out_dir_jadax.mkdir(parents=True, exist_ok=True)
    out_dir_apktool.mkdir(parents=True, exist_ok=True)
    out_dir_mariana.mkdir(parents=True, exist_ok=True)

    print("[*] Starting parallel extraction (JADX and APKTool)...")

    with ThreadPoolExecutor(max_workers=2) as executor:
        futures = [
            executor.submit(run_jadx, apk_path, str(out_dir_jadax), work_dir),
            executor.submit(run_apktool, apk_path, str(out_dir_apktool))
        ]
        for future in futures:
            future.result()

    print("[+] Extraction completed. Starting static analysis with Mariana Trench...")
    result = run_mariana_trench(apk_path, str(out_dir_mariana), str(out_dir_jadax), work_dir)
    if result.returncode != 0:
        print(f"\n[-] Mariana Trench failed with exit code {result.returncode}")
        print("====== STDOUT ======")
        print(result.stdout)
        print("====== STDERR ======")
        print(result.stderr)
        print("======================")
print("[+] Stage1 successfully completed!")
```

Скрипт stage2.py

```
import os
import re
import json
import subprocess
from concurrent.futures import ThreadPoolExecutor
from pathlib import Path
```

```
import dex_tracer

def run_regex(sources_dir, out_file):
    REGEX_RULES = {
    # --- CREDENTIALS, SECRETS & TOKENS ---
    "AWS_API_Key": r"AKIA[0-9A-Z]{16}",
    "AWS_Secret_Key": r"[0-9a-zA-Z+/]{40}",
    "Google_API_Key": r"AIza[0-9A-Za-z\\-_]{35}",
    "Firebase_URL": r"https://[a-zA-Z0-9\\-_]+\\.firebaseio\\.com",
    "GitHub_PAT": r"ghp_[a-zA-Z0-9]{36}",
      "Slack_Webhook":  r"https://hooks\\.slack\\.com/services/T[a-zA-Z0-9_]+/B[a-zA-Z0-9_]+/[a-zA-
Z0-9_]+",
    "JWT_Token": r"ey[A-Za-z0-9\-_=]+\.ey[A-Za-z0-9\-_=]+\.[A-Za-z0-9\-_=]+",
     "Generic_Secret_or_Password": r"(?i)(password|secret|passwd|token|private_key|auth_token|api_se-
cret)\s*=\s*['\"][A-Za-z0-9+/=_\-]{8,}['\"]",
    "RSA_Private_Key_Header": r"-----BEGIN[^-\r\n]+PRIVATE KEY-----",

    # --- NETWORK & INSECURE INFRASTRUCTURE ---
    "Insecure_HTTP_Protocol": r"\bhttp://[a-zA-Z0-9\\-.&_\\/=?:]+",
     "Internal_IP_Target": r"\b(192\.168\.\d{1,3}\.\d{1,3}|10\.\d{1,3}\.\d{1,3}\.\d{1,3}|172\.(1[6-9]|2\d|
3[01])\.\d{1,3}\.\d{1,3})\b",
    "Bind_All_Interfaces": r"\b0\.0\.0\.0\b",

    # --- WEAK CRYPTOGRAPHY ---
    "Weak_Hash_MD5": r"(?i)MessageDigest\.getInstance\(\s*['\"]MD5['\"]",
    "Weak_Hash_SHA1": r"(?i)MessageDigest\.getInstance\(\s*['\"]SHA-1['\"]",
    "Insecure_Cipher_AES_ECB": r"(?i)AES/ECB/(PKCS5Padding|NoPadding)",

    # --- INFORMATION LEAKAGE & DIAGNOSTICS ---
    "Android_Log_Leak": r"\bLog\.(e|d|v|i|wtf)\s*\(",
    "Java_PrintStackTrace": r"\.printStackTrace\s*\(\s*\)",
    "System_Out_Err_Print": r"\bSystem\.(out|err)\.print",

    # --- ANDROID MANIFEST CONFIGURATION FLAWS ---
    "Manifest_AllowBackup_Enabled": r"android:allowBackup\s*=\s*['\"]true['\"]",
    "Manifest_Debuggable_Enabled": r"android:debuggable\s*=\s*['\"]true['\"]",
    "Manifest_CleartextTraffic_Allowed": r"android:usesCleartextTraffic\s*=\s*['\"]true['\"]",
    "Manifest_ExportedComponent_Unprotected": r"android:exported\s*=\s*['\"]true['\"]"
}
    print("[*] Starting Regex Pattern Matching...")
    findings = []

    for root, _, files in os.walk(sources_dir):
        for file in files:
            if file.endswith(".java") or file == "AndroidManifest.xml":
                file_path = os.path.join(root, file)
                try:
                    with open(file_path, "r", encoding="utf-8", errors="ignore") as f:
                        for line_num, line in enumerate(f, 1):
                            clean_line = line.strip()
                            for rule_name, pattern in REGEX_RULES.items():
                                matches = re.finditer(pattern, clean_line)
```

```
                    for match in matches:
                        findings.append({
                            "rule": rule_name,
                            "file": os.path.relpath(file_path, sources_dir),
                            "line": line_num,
                            "match": match.group(0)[:150]
                        })
            except Exception as e:
                pass

    with open(out_file, "w", encoding="utf-8") as f:
        json.dump(findings, f, indent=4, ensure_ascii=False)

    print(f"[+] Regex scan completed. Found {len(findings)} potential signature matches.")



def run_semgrep(sources_dir, out_file, work_dir):
    print("[*] Starting Semgrep AST Analysis...")
    cmd = f"semgrep scan -c {work_dir}/tools/rules/ --json -o {out_file} {sources_dir}"

        result = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE,
text=True)

    if result.returncode not in [0, 1]:
        print(f"[-] Semgrep failed: {result.stderr}")
    else:
        print("[+] Semgrep AST Analysis completed.")

def run_stage_2(apk_path, work_dir, out_dir, jadx_sources_dir):
    print("\n" + "="*40)
    print("[+] STARTING STAGE 2: PARALLEL STATIC ANALYSIS")
    print("="*40)

    regex_report = out_dir / "regex_results.json"
    semgrep_report = out_dir / "semgrep_results.json"
    tracing_report = out_dir / "call_chains.json"
    with ThreadPoolExecutor(max_workers=3) as executor:
        executor.submit(run_regex, jadx_sources_dir, str(regex_report))
        executor.submit(run_semgrep, jadx_sources_dir, str(semgrep_report), work_dir)

    print("[+] Correlating Semgrep findings with Dex Call Graph...")
    dex_tracer.run_reachability_analysis(
        apk_path=apk_path,
        semgrep_report_path=str(semgrep_report),
        output_report_path=str(tracing_report)
    )
    print("\n[+] STAGE 2 SUCCESSFULLY COMPLETED!")
print(f"[+] All reports saved to: {out_dir}\n")
```

Файл з правилами для mariana trench rules.json


```
[
  {
    "name": "Third-party flows into code execution sink (RCE)",
    "code": 1,
     "description": "Values from third-party controlled source may eventually flow into code execution sink",
    "sources": [
      "ActivityUserInput",
      "AndroidxNavigationUserInput",
      "ProviderUserInput",
      "ReceiverUserInput",
      "ServiceUserInput"
    ],
    "sinks": [
      "CodeExecution"
    ]
  },
  {
    "name": "Third-party input flows into file resolver",
    "code": 2,
     "description": "Values from third-party controlled source may eventually flow into sink file resolver",
    "sources": [
      "ActivityUserInput",
      "AndroidxNavigationUserInput",
      "ProviderUserInput",
      "ReceiverUserInput",
      "ServiceUserInput"
    ],
    "sinks": [
      "FileResolver",
      "InputStream"
    ]
  },
  {
    "name": "Third-party flows into implicitly launched intent",
    "code": 3,
    "description": "Values from third-party controlled source may eventually flow into an implicit intent and intent launcher",
    "multi_sources": {
      "a": [
        "IntentCreation"
      ],
      "b": [
        "ActivityUserInput",
        "AndroidxNavigationUserInput",
        "ReceiverUserInput",
        "ServiceUserInput"
      ]
    },
    "partial_sinks": [
```

```
      "LaunchingComponent"
    ]
  },
  {
    "name": "Third-party flows into raw SQL statement",
    "code": 4,
    "description": "Values from third-party controlled source may eventually flow into a raw SQL statement potentially causing SQL injection",
    "sources": [
      "ActivityUserInput",
      "AndroidxNavigationUserInput",
      "ProviderUserInput",
      "ReceiverUserInput",
      "ServiceUserInput"
    ],
    "sinks": [
      "SQLQuery"
    ]
  },
  {
    "name": "Third-party input flows into WebView Javascript execution API",
    "code": 5,
     "description": "Values from third-party controlled source may eventually flow into a Webview Javascript execution API",
    "sources": [
      "ActivityUserInput",
      "AndroidxNavigationUserInput",
      "ProviderUserInput",
      "ReceiverUserInput",
      "ServiceUserInput"
    ],
    "sinks": [
      "ExecuteJavascript"
    ]
  },
  {
    "name": "Exported component calls setResult",
    "code": 6,
    "description": "Detect call chain flows from Activity lifecycle methods to setResult",
    "sources": [
      "ActivityLifecycle"
    ],
    "sinks": [
      "SetResult"
    ]
  }
]
```

Файл з конфігурацією лайфтаймів для mariana trench lifecycles.json


```
[
  {
   "base_class_name": "Landroidx/fragment/app/FragmentActivity;",
   "method_name": "fragment_activity_lifecycle_wrapper",
   "callees": [
     {
      "method_name": "onCreate",
      "return_type": "V",
      "argument_types": [
       "Landroid/os/Bundle;"
      ]
     },
     {
      "method_name": "onStart",
      "return_type": "V",
      "argument_types": []
     },
     {
      "method_name": "onResume",
      "return_type": "V",
      "argument_types": []
     },
     {
      "method_name": "onPause",
      "return_type": "V",
      "argument_types": []
     },
     {
      "method_name": "onStop",
      "return_type": "V",
      "argument_types": []
     },
     {
      "method_name": "onRestart",
      "return_type": "V",
      "argument_types": []
     },
     {
      "method_name": "onDestroy",
      "return_type": "V",
      "argument_types": []
     }
   ]
  },
  {
   "base_class_name": "Landroidx/fragment/app/Fragment;",
   "method_name": "xfragment_lifecycle_wrapper",
   "callees": [
     {
      "method_name": "onCreate",
      "return_type": "V",
```

```
  "argument_types": [
   "Landroid/os/Bundle;"
  ]
 },
 {
  "method_name": "onCreateView",
  "return_type": "V",
  "argument_types": [
   "Landroid/view/LayoutInflater;",
   "Landroid/view/ViewGroup;",
   "Landroid/os/Bundle;"
  ]
 },
 {
  "method_name": "onCreateView",
  "return_type": "V",
  "argument_types": [
   "Landroid/view/LayoutInflater;",
   "Landroid/view/ViewGroup;",
   "Landroid/os/Bundle;"
  ]
 },
 {
  "method_name": "onViewCreated",
  "return_type": "V",
  "argument_types": [
   "Landroid/view/View;",
   "Landroid/os/Bundle;"
  ]
 },
 {
  "method_name": "onViewStateRestored",
  "return_type": "V",
  "argument_types": [
   "Landroid/os/Bundle;"
  ]
 },
 {
  "method_name": "onStart",
  "return_type": "V",
  "argument_types": []
 },
 {
  "method_name": "onResume",
  "return_type": "V",
  "argument_types": []
 },
 {
  "method_name": "onPause",
  "return_type": "V",
  "argument_types": []
 },
```

```
    {
     "method_name": "onStop",
     "return_type": "V",
     "argument_types": []
    },
    {
     "method_name": "onSaveInstanceState",
     "return_type": "V",
     "argument_types": [
      "Landroid/os/Bundle;"
     ]
    },
    {
     "method_name": "onDestroyView",
     "return_type": "V",
     "argument_types": []
    },
    {
     "method_name": "onDestroy",
     "return_type": "V",
     "argument_types": []
    }
   ]
  },
  {
   "base_class_name": "Landroid/app/Fragment;",
   "method_name": "fragment_lifecycle_wrapper",
   "callees": [
    {
     "method_name": "onCreate",
     "return_type": "V",
     "argument_types": [
      "Landroid/os/Bundle;"
     ]
    },
    {
     "method_name": "onCreateView",
     "return_type": "V",
     "argument_types": [
      "Landroid/view/LayoutInflater;",
      "Landroid/view/ViewGroup;",
      "Landroid/os/Bundle;"
     ]
    },
    {
     "method_name": "onCreateView",
     "return_type": "V",
     "argument_types": [
      "Landroid/view/LayoutInflater;",
      "Landroid/view/ViewGroup;",
      "Landroid/os/Bundle;"
     ]
```

```
},
{
 "method_name": "onViewCreated",
 "return_type": "V",
 "argument_types": [
  "Landroid/view/View;",
  "Landroid/os/Bundle;"
 ]
},
{
 "method_name": "onViewStateRestored",
 "return_type": "V",
 "argument_types": [
  "Landroid/os/Bundle;"
 ]
},
{
 "method_name": "onStart",
 "return_type": "V",
 "argument_types": []
},
{
 "method_name": "onResume",
 "return_type": "V",
 "argument_types": []
},
{
 "method_name": "onPause",
 "return_type": "V",
 "argument_types": []
},
{
 "method_name": "onStop",
 "return_type": "V",
 "argument_types": []
},
{
 "method_name": "onSaveInstanceState",
 "return_type": "V",
 "argument_types": [
  "Landroid/os/Bundle;"
 ]
},
{
 "method_name": "onDestroyView",
 "return_type": "V",
 "argument_types": []
},
{
 "method_name": "onDestroy",
 "return_type": "V",
 "argument_types": []
```

```
    }
   ]
  },
  {
   "base_class_name": "Landroid/app/Activity;",
   "method_name": "activity_lifecycle_wrapper",
   "callees": [
    {
     "method_name": "onCreate",
     "return_type": "V",
     "argument_types": [
      "Landroid/os/Bundle;"
     ]
    },
    {
     "method_name": "onStart",
     "return_type": "V",
     "argument_types": []
    },
    {
     "method_name": "onResume",
     "return_type": "V",
     "argument_types": []
    },
    {
     "method_name": "onPause",
     "return_type": "V",
     "argument_types": []
    },
    {
     "method_name": "onStop",
     "return_type": "V",
     "argument_types": []
    },
    {
     "method_name": "onRestart",
     "return_type": "V",
     "argument_types": []
    },
    {
     "method_name": "onDestroy",
     "return_type": "V",
     "argument_types": []
    }
   ]
  },
  {
   "base_class_name": "Landroid/app/Service;",
   "method_name": "service_lifecycle_wrapper",
   "callees": [
    {
     "method_name": "onCreate",
```

```
      "return_type": "V",
      "argument_types": []
    },
    {
      "method_name": "onStartCommand",
      "return_type": "I",
      "argument_types": [
        "Landroid/content/Intent;",
        "I",
        "I"
      ]
    },
    {
      "method_name": "onBind",
      "return_type": "Landroid/os/IBinder;",
      "argument_types": [
        "Landroid/content/Intent;"
      ]
    },
    {
      "method_name": "onUnbind",
      "return_type": "Z",
      "argument_types": [
        "Landroid/content/Intent;"
      ]
    },
    {
      "method_name": "onRebind",
      "return_type": "V",
      "argument_types": [
        "Landroid/content/Intent;"
      ]
    },
    {
      "method_name": "onDestroy",
      "return_type": "V",
      "argument_types": []
    }
  ]
 }
]
```

Файл з конфігурацією для mariana trench default_generator_config.json

```
[
  {
   "name": "ActivityResultGenerator"
  },
  {
   "name": "ActivitySourceGenerator"
  },
  {
   "name": "AndroidBroadcastSourceGenerator"
  },
  {
   "name": "AndroidIntentLaunchSinkGenerator"
  },
  {
   "name": "AndroidxNavigationSourceGenerator"
  },
  {
   "name": "CodeExecutionSinkGenerator"
  },
  {
   "name": "CommonSanitizers"
  },
  {
   "name": "content_provider_generator"
  },
  {
   "name": "DataCastFeatureGenerator"
  },
  {
   "name": "ExplicitIntentFeatureGenerator"
  },
  {
   "name": "FileSinkGenerator"
  },
  {
   "name": "ImplicitIntentSourceGenerator"
  },
  {
   "name": "IntentDataFeatureGenerator"
  },
  {
   "name": "IntentUtilsChooserSanitizer"
  },
  {
   "name": "join_override_generator"
  },
  {
   "name": "ReflectionGenerator"
  },
  {
```

```
    "name": "SensitiveCookieDataGenerator"
  },
  {
    "name": "ServiceSourceGenerator"
  },
  {
    "name": "service_sources"
  },
  {
    "name": "SQLiteDatabaseSinksGenerator"
  },
  {
    "name": "taint_in_taint_out"
  },
  {
    "name": "taint_in_taint_this"
  },
  {
    "name": "WebViewSinkGenerator"
  }
]
```

Файл з конфігурацією шляхів для mariana trench default_generator_config.json

```
[
  "/mnt/c/Users/Mor/diploma/tools/AndroidSDK/platforms"
]
```

Файл правил network-security-config.yml для semgrep

```
rules:
- id: nsc-allows-plaintext-traffic
  languages:
  - generic
  message: >-
    The Network Security Config is set to allow non-encrypted connections.
    Evaluate if this is necessary for your app, and disable it if appropriate.
    (To hide this warning, set `xmlns:tools="http://schemas.android.com/tools"
    tools:ignore="InsecureBaseConfiguration"` as parameters to your
    `<network-security-config>`)
  metadata:
    category: best-practice
    technology:
    - android
    references:
    - https://developer.android.com/training/articles/security-config
          - https://www.nowsecure.com/blog/2018/08/15/a-security-analysts-guide-to-network-security-
configuration-in-android-p/
  patterns:
  - pattern: |
      <base-config ... cleartextTrafficPermitted="true" ... >
  - pattern-not-inside: |
      <!-- ... -->
     # If the config explicitly tells us not to check for insecure configurations, respect that
      # (on a best-effort basis due to limitations of how much you can glob in generic parser mode)
```

```
  - pattern-not-inside: |
      <network-security-config ... InsecureBaseConfiguration ... >... ... ... ... ... ... ... ... ... ... </network-
security-config>
  severity: INFO
  paths:
    include:
    - '*.xml'
- id: nsc-pinning-without-backup
  languages:
  - generic
  message: >-
    Your app uses TLS public key pinning without specifying a backup key.
    If you are forced to change TLS keys or CAs on short notice, not
    having a backup pin can lead to connectivity issues until you can push
    out an update. It is considered best practice to add at least one additional
    pin as a backup.
  metadata:
    category: best-practice
    technology:
    - android
    references:
    - https://developer.android.com/training/articles/security-config#CertificatePinning
          - https://www.nowsecure.com/blog/2018/08/15/a-security-analysts-guide-to-network-security-
configuration-in-android-p/
  patterns:
      # FIXME: This check will currently not detect cases where there are two pins
      # listed, but one of them is inside a <!-- comment --> - these will be recognized
      # as having two or more pins. I don't think detecting these cases while not falsely
      # detecting cases where there are three pins, but the middle one is commented out,
      # is possible using the generic parser - this would require a specialized XML parser
      # that has knowledge about comments etc.
  - pattern: |
      <pin ...>...</pin>
  - pattern-not-inside: |
      <pin ...>...</pin>...<pin ...>...</pin>
  - pattern-inside: |
      <pin-set ...> ... ... </pin-set>
  - pattern-inside: |
      <domain-config ... > ... ... ... ... ... </domain-config>
  - pattern-not-inside: |
      <!-- ... -->
  severity: INFO
  paths:
    include:
    - '*.xml'
- id: nsc-pinning-without-expiration
  languages:
  - generic
  message: >-
    Your app uses TLS public key pinning without specifying an expiration date.
    If your users do not update the app to receive new pins in time, expired or replaced
    certificates can lead to connectivity issues until they install an update.
```

```
    It is considered best practice to set an expiration time, after which the system will
    default to trusting system CAs and disregard the pin.
  metadata:
    category: best-practice
    technology:
    - android
    references:
    - https://developer.android.com/training/articles/security-config#CertificatePinning
          - https://www.nowsecure.com/blog/2018/08/15/a-security-analysts-guide-to-network-security-
configuration-in-android-p/
  patterns:
  - pattern: |
      <pin-set ...>... ... ...</pin-set>
  - pattern-not-inside: |
      <pin-set ... expiration="..."> ... ... ... </pin-set>
  - pattern-inside: |
      <domain-config ... > ... ... ... ... ... </domain-config>
  - pattern-not-inside: |
      <!-- ... -->
  severity: INFO
  paths:
    include:
    - '*.xml'
- id: nsc-allows-user-ca-certs
  languages:
  - generic
  message: >-
    The Network Security Config is set to accept user-installed CAs.
    Evaluate if this is necessary for your app, and disable it if appropriate.
    (To hide this warning, set `xmlns:tools="http://schemas.android.com/tools"
    tools:ignore="AcceptsUserCertificates"` as parameters to your
    `<network-security-config>`)
  metadata:
    category: best-practice
    technology:
    - android
    references:
    - https://developer.android.com/training/articles/security-config
          - https://www.nowsecure.com/blog/2018/08/15/a-security-analysts-guide-to-network-security-
configuration-in-android-p/
  patterns:
  - pattern: |
      <certificates ... user ... />
  - pattern-inside: |
      <base-config ... > ... ... ... ... </base-config>
  - pattern-not-inside: |
      <!-- ... -->
     # If the config explicitly tells us not to check for user CAs, respect that
      # (on a best-effort basis due to limitations of how much you can glob in generic parser mode)
  - pattern-not-inside: |
      <network-security-config ... AcceptsUserCertificates ... >... ... ... ... ... ... ... ... ... ... </network-secu-
rity-config>
```

```
  severity: WARNING
  paths:
    include:
    - '*.xml'
- id: nsc-allows-user-ca-certs-for-domain
  languages:
  - generic
  message: >-
    The Network Security Config is set to accept user-installed CAs for the
    domain `$DOMAIN`.
    Evaluate if this is necessary for your app, and disable it if appropriate.
    (To hide this warning, set `xmlns:tools="http://schemas.android.com/tools"
    tools:ignore="AcceptsUserCertificates"` as parameters to your
    `<network-security-config>`)
  metadata:
    category: best-practice
    technology:
    - android
    references:
    - https://developer.android.com/training/articles/security-config
          - https://www.nowsecure.com/blog/2018/08/15/a-security-analysts-guide-to-network-security-
configuration-in-android-p/
  patterns:
  - pattern: |
      <certificates src="user" ... />
  - pattern-inside: |
      <trust-anchors> ... ... ... </trust-anchors>
  - pattern-inside: |
      <domain-config ... > ... <domain ...> $DOMAIN </domain>... ... ... </domain-config>
  - pattern-not-inside: |
      <!-- ... -->
     # If the config explicitly tells us not to check for user CAs, respect that
      # (on a best-effort basis due to limitations of how much you can glob in generic parser mode)
  - pattern-not-inside: |
      <network-security-config ... AcceptsUserCertificates ... >... ... ... ... ... ... ... ... ... ... </network-secu-
rity-config>
  severity: WARNING
  paths:
    include:
- '*.xml'
```

Файл правил exported_activity.yaml для semgrep

```
- id: exported_activity
  patterns:
  - pattern-not-inside: <activity ... android:exported="false" ... />
  - pattern-inside: "<activity  ... /> \n"
  - pattern-either:
    - pattern: |
        <activity ... android:exported="true" ... />
    - pattern: |
        <activity ... <intent-filter> ... />
  message: >-
```

The application exports an activity. Any application on the device can launch the exported activity which may compromise the integrity of your application or its data.

Ensure that any exported activities do not have privileged access to your application's control plane.

```
 languages:
 - generic
 severity: WARNING
 paths:
   exclude:
   - sources/
   - classes3.dex
   - '*.so'
   include:
   - '*AndroidManifest.xml'
 metadata:
   category: security
   subcategory:
   - vuln
   cwe:
   - 'CWE-926: Improper Export of Android Application Components'
   confidence: MEDIUM
   likelihood: MEDIUM
   impact: MEDIUM
   owasp:
   - A5:2021 Security Misconfiguration
   technology:
   - Android
   references:
- https://cwe.mitre.org/data/definitions/926.html
```

Файл правил insecure_cipher_3des.yml для semgrep

```
rules:
  - id: insecure_cipher_3des
    severity: WARNING
    languages:
      - java
      - kotlin
    metadata:
      category: security
      confidence: HIGH
      likelihood: HIGH
      impact: HIGH
      subcategory:
        - vuln
      owasp-mas:
        identifier: MASVS-CRYPTO-1
        area: cryptography
        level:
          - L1
          - L2
      references:
            - https://mas.owasp.org/MASTG/General/0x04g-Testing-Cryptography/#identifying-insecure-andor-deprecated-cryptographic-algorithms
```

-
https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html#cipher-modes
    message: >-
      This Android mobile application seems using a weak cipher, `3DES`, which could expose to various kind of vulnerabilities.
      The data encrypted using `3DES` mode block could allow to retrieve its plain-text value.
      It is recommended to avoid the use of `3DES`, and use `AES` instead.
    pattern-either:
      - pattern: Cipher $VAR = Cipher.getInstance("=~/((3DES|DESede|DESEDEWRAP)\/.*\/.*|(3DES|DESede|DESEDEWRAP))/i", ...);
      - pattern: SecretKeyFactory $VAR = SecretKeyFactory.getInstance("=~/(DESede\/.*\/.*|DESede)$/i", ...);
      - pattern: $VAR = Cipher.getInstance("=~/((3DES|DESede|DESEDEWRAP)\/.*\/.*|(3DES|DESede|DESEDEWRAP))/i", ...);
- pattern: $VAR = SecretKeyFactory.getInstance("=~/(DESede\/.*\/.*|DESede)$/i", ...);

Файл правил unsecure_exported_activity.yml для semgrep

```
rules:
  - id: unsecure_exported_activity
    severity: WARNING
    languages:
      - xml
    metadata:
      category: security
      confidence: HIGH
      likelihood: MEDIUM
      impact: LOW
      subcategory:
        - vuln
      owasp-mas:
        identifier: MASVS-PLATFORM-1
        area: platform
        level:
          - L1
          - L2
      references:
        - https://mas.owasp.org/MASTG/tests/android/MASVS-PLATFORM/MASTG-TEST-0029/
    message: >-
      This Android mobile application seems to be exporting an activity. Any application on the device can launch the exported activity which may compromise the integrity of your application or its data.
      Ensure that any exported activities do not have privileged access to your application's control plane.
    patterns:
      - pattern-not-inside: |
          <activity ... android:exported="false" ... />
      - pattern-either:
        - pattern: |
            <activity ... android:exported="true" ... />
        - pattern: |
<activity ... > ... <intent-filter> ... </activity>
```

Файл правил device_access_security_policy.yml для semgrep

```
rules:
  - id: device_access_security_policy
    severity: WARNING
    languages:
      - xml
    metadata:
      category: security
      confidence: LOW
      likelihood: LOW
      impact: LOW
      subcategory:
        - vuln
      owasp-mas:
        identifier: MASVS-STORAGE-1
        area: storage
        level:
          - L2
      references:
        - https://mas.owasp.org/MASTG/tests/android/MASVS-STORAGE/MASTG-TEST-0012
    message: >-
       This Android mobile application seems to be targeting a API level earlier than 23, which could
expose to various kind of vulnerabilities.
      Please up use the a newer version.
    patterns:
      - pattern: <uses-sdk android:minSdkVersion="$X" />
      - metavariable-comparison:
comparison: int($X)<23
```