НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
Завідувач кафедри

_____ **Дмитро ЛАНДЕ**

(підпис)

«_____»_____2022 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системи, технології та математичні методи кібербезпеки»
спеціальності 125 «Кібербезпека»

на тему: Методи захисту ОС Adroid від атак Code Execution

Виконав (-ла): здобувач вищої освіти **IV** курсу, групи ФБ-83

Бондарчук Ярослава Юріївна
(прізвище, ім'я, по батькові) (підпис)

Керівник: Барановський Олексій Миколайович, к.т.н, доцент
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент:
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Здобувач вищої освіти _____
(підпис)

Київ – 2022 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

«___»_____2022 р.

ЗАВДАННЯ
на дипломну роботу здобувачу вищої освіти

Бондарчук Ярослава Юріївна
(прізвище, ім'я, по батькові)

1. Тема роботи: Методи захисту ОС Adroid від атак Code Execution
керівник роботи Барановський Олексій Миколайович, к.т.н, доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « » _____ 2022 р. №

2. Термін подання здобувачем вищої освіти роботи 14 червня 2022 р.

3. Вихідні дані до роботи: Існуючі методи захисту ОС Adroid від атак Code Execution

4. Зміст роботи: огляд безпекової моделі ОС Android, аналіз даних вразливостей типу Code Execution в ОС Android за 2022 рік, аналіз існуючих методів захисту ОС Android від RCE атак, опис та тестування додаткових методів захисту від вразливості типу Code Execution.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): презентація

6. Дата видачі завдання 15 листопада 2021 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Визначення напрямку роботи, постановка задачі	20.09.2021-15.10.2021	
2	Вибір теми роботи, формування основних завдань та мети	16.10.2021-20.11.2021	
3	Визначення рекомендованої наукової літератури та її опрацювання	21.11.2021-15.12.2021	
4	Робота над першим розділом; аналіз літературних джерел	16.12.2021-15.02.2022	
5	Робота над другим розділом	16.02.2022 - 15.03.2022	
6	Написання третього розділу	16.03.2022 - 15.04.2022	
7	Робота над четвертим розділом	16.04.2022 - 29.05.2022	
8	Проходження переддипломної практики	02.05.2022 - 29.05.2022	
9	Тестування запропонованих методів захисту ОС Android	30.05.2022-05.06.2022	
10	Створення презентації для передзахисту дипломної роботи	06.06.2022-12.06.2022	
11	Передзахист дипломної роботи	13.06.2022	
12	Доопрацювання	14.06.2022-19.06.2022	
13	Захист дипломної роботи	20.06.2022	

Здобувач вищої освіти _____ Ярослава БОНДАРЧУК

(підпис) (**Власне ім'я, ПРІЗВИЩЕ**)

Керівник роботи _____ Олексій БАРАНОВСЬКИЙ

(підпис) (**Власне ім'я, ПРІЗВИЩЕ**)

РЕФЕРАТ

Дипломна робота має обсяг 57 сторінок, містить 28 рисунків, 3 таблиці та 16 джерел посилань.

Об’єкт роботи: методи захисту безпеки ОС Android

Предмет дослідження: методи запобігання Code Execution вразливостям ОС Android

Метою роботи: аналіз безпекової моделі ОС Android та методів захисту від RCE атак та пропозиція методу для запобігання вразливостей типу Code Execution.

Методи дослідження**:** опрацювання літератури щодо обраної теми, проведення аналізу поточних методів захисту ОС Android від Code Execution вразливостей, тестування запропонованих методів покращення безпеки ОС Android.

Практичне значення одержаних результатів полягає у формуванні та реалізації методу захисту для запобігання Code Execution вразливостям в ОС Android.

Ключові слова: вразливість, Android, remote code execution, пам’ять

ABSTRACT

Thesis has a volume of 57 pages, contains 28 figures, 3 tables and 16 sources of references.2

The object of study are Android security protection methods.

The subject of the study are methods to prevent Code Execution vulnerabilities in Android.

The aim of the study is to make analys of the security model of Android OS and methods of protection against RCE attacks and a propose a method to prevent Code Execution vulnerabilities.

The research methods are to study the literature on the selected topic, to analyze the current methods of protecting Android OS from Code Execution of vulnerabilities, testing the proposed methods to improve the security of Android OS.

The practical significance of the work is to implement a method of protection to prevent Code Execution vulnerabilities in Android OS.

Key words: vulnerability, Android, remote code execution, memory

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів…… 7

Вступ……8

1 Огляд вразливостей ОС Android — RCE атаки та їх класифікація……... 10

1.1 Архітектура ОС Android…… 10

1.2 Модель безпеки ОС Android……..11

1.3 Огляд вразливостей ОС Андроїд……..13

1.4 Remote Code Execution атаки та їх типи……...15

Висновки до розділу 1……...25

2 Опис та аналіз існуючих методів підвищення рівня захищеності ОС Android…27

2.1 Методи захисту від десеріалізації……27

2.2 Методи захисту від атак переповнення буфера…….. 28

2.3 Методи захисту від атак невідповідності типів……...33

Висновки до розділу 2……...34

3. Опис запропонованих методів для запобігання Code Execution вразливостей... 35

3.1 Morula…….. 35

3.2 SSPFA……...39

Висновки до розділу 3……...42

4 Реалізація……43

4.1 Технічний опис системи…… 43

4.2 Аналіз ефективності застосування методів Morula та SSPFA……45

4.3 Аналіз впливу методів на продуктивність ОС Android……...49

Висновки до розділу 4……...52

Висновки……54

Перелік джерел посилань……..55

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ОС —- операційна система

RCE — remote code execution

ASLR — Address Space Layout Randomization

DEP — Data Execution Prevention

SSP — Stack Smashing Protector

ВСТУП

Актуальність. У перші дні єдиною метою Android було працювати на смартфонах. Але завдяки подальшому вдосконаленню ОС знайшла застосування в смарт-телевізорах, смарт-годинниках, медичному обладнанні тощо. На травень 2022 року 71,45%[1] припадає на пристрої з ОС Android. Дана ОС активно використовуються в багатьох сферах, таких як банківська справа, платежі, транзакції та соціальні мережі. З таким широким використанням і залежностями безпека систем Android стала серйозною проблемою.

Відкритість Android створює сприятливу атмосферу як для користувачів, розробників, а також вона приваблює зловмисників і хакерів, щоб скористатися неправомірними перевагами. За даними Google з 2009 року було знайдено 4275 вразливості в ОС Android, найбільша кількість типу Code Execution - 17%. Тому покращення методів захисту від атак RCE є актуальною темою.

Об’єктом роботи є методи захисту ОС Android

Предметом дослідження методи запобігання Code Execution вразливостям ОС Android

Метою роботи є аналіз безпекової моделі ОС Android, методів захисту від RCE атак та пропозиція методу для запобігання вразливостей типу Code Execution.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

1. Розглянути безпекову модель ОС Android
2. Проаналізувати дані про вразливості типу Code Execution в ОС Android за 2022 рік
3. Навести та проаналізувати існуючі методи захисту ОС Android від RCE атак.
4. Запропонувати та протестувати додаткові методи захисту від вразливості типу Code Execution

Методи дослідження: опрацювання літератури щодо обраної теми, проведення аналізу поточних методів захисту ОС Android від Code Execution вразливостей, тестування запропонованих методів покращення безпеки ОС Android.

Практичне значення одержаних результатів полягає у формуванні та реалізації методу захисту для запобігання Code Execution вразливостям в ОС Android.

1 ОГЛЯД ВРАЗЛИВОСТЕЙ ОС ANDROID — RCE АТАКИ ТА ЇХ КЛАСИФІКАЦІЯ

1.1 Архітектура ОС Android

Архітектура системи Android має чотири основні рівні, які надають послуги для рівня вище та використовують послуги рівня нижче. Android складається з віртуальної машини DVM та основних бібліотек Java. Нативні бібліотеки Android написані на C або C++ , компілюються за допомогою певної апаратної архітектури, яка використовується пристроєм, і лежать над ядром Linux.

Додатки для Android можна вважати основним джерелом атаки. Наприклад витоку конфіденційних або приватних даних - контакти, повідомлення, місцезнаходження тощо.

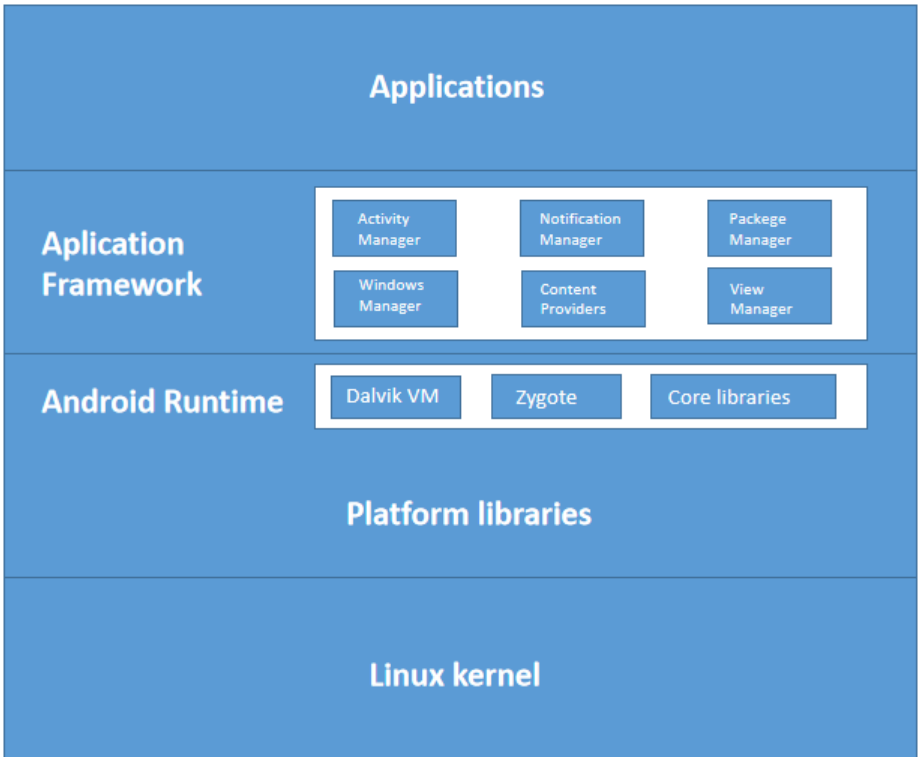

Рисунок 1.1 — Архітектура ОС Android

1.2 Модель безпеки ОС Android[2]

Android в першу чергу орієнтований на кінцевого користувача, система за замовчуванням має бути захищеною. Користувач очікує, що інші сторони вжили необхідних заходів для забезпечення його безпеки. Однак в даній ОС, характер дії вимагає його згоди, кінцевий користувач є вирішальним фактором, коли мова йде про безпеку всієї системи. Розглянемо рішення безпеки, яке пропонує нам Android.

Безпекова модель ОС Android складається з трьох частин: операційна система, додатки, користувач.

1.2.1. Операційна система

За безпеку ОС відповідає в першу чергу ядро, яке є основним програмним забезпеченням операційної системи, що обробляє ресурси ЦП, системну пам'ять, системні пристрої, включаючи файлові системи та мережу та відповідає за управління всіма процесами. Воно є сполучною ланкою між програмним та апаратним забезпеченням. Ядро Android засноване на гілках довгострокової підтримки (LTS) ядра Linux. Безпека операційної системи Android базується на таких ключових функціях безпеки ядра Linux: ізоляція процесу, модель дозволів на основі користувачів, міжпроцесний зв'язок (IPC).

Процес пісочниці є другим ключовим засобом безпеки від ОС. Дана техніка використовується для розділення програми та системного ресурсу. Завдяки пісочниці кожна програма має унікальний ідентифікатор для доступу до власного файлу і також до іншого файлу, який має відповідний маркер читання/запис/виконання для іншої програми.

Рутування — звичайний користувач не має повних прав у системі, дане дизайнерське рішення було прийняте з міркувань безпеки. Через те що зазвичай

користувачами Android є звичайні люди, що знаються на технологіях лише поверхнево та часто навіть не читають запити дозволів від завантажуваних додатків. Але так як система Android є відкритою, користувач може отримати root права, та після цього не можна вже покладатися на заходи безпеки, запроваджені операційною системою.

Перевірене завантаження — процес, що гарантує завантаження оригінальної версії ОС Android з аферентним системним кодом. Verified Boot встановлює ланцюг довіри між кількома компонентами, які можна змінити, починаючи від апаратного забезпечення і закінчуючи перевіреними розділами. Якщо будь-який з компонентів вище ланцюга змінено, весь ланцюжок стає недійсним, а користувач отримує попередження. При цьому отримуємо два стани пристрою: заблоковано та розблоковано. При першому проходить активна перевірка завантаження та не можна прошити користувацьке ПЗ, при наступному можна повноцінно користуватися системою. При даному процесі також використовується ключ для підпису копії Android, яка працює на пристрої, він є частиною перевірки завантаження. Також існує захист відкату від намагання зловмисника понизити версію Android,він є частиною процесу підтвердженого завантаження.

1.2.2 Додатки

Аспект безпеки додатків досить важливий, так як саме вони є вектором атаки зловмисника.

Механізм дозволів - захищає конфіденційність користувача Android. Згода користувача на виконання дій, дає можливість впливати на інші програми, операційну систему або самого користувача. Дозволи, необхідні для програми, оголошуються в AndroidManifest.xml, кожен з яких вказується у власному тегу uses-permission.Деякі дозволи надаються програмі за замовчуванням, проте існує

інша категорія, вимагає спеціальної згоди користувача. Починаючи з Android 6.0 програми почали запитувати дозволи під час свого виконання пропонуючи діалогове вікно користувачу перед його дією. Раніше дозволи потрібно було прийняти при завантаженні.

Технології зберігання даних є досить важливими для системи, так як вони є кінцевою метою зловмисника. Android пропонує три варіанти збереження даних: внутрішня та зовнішня пам’ять, абстракція від постачальників контенту. Якщо кілька програм розроблені під одним підписом, можна бмінюватися даними між ними за допомогою постачальника контенту, щоб дані були видимі лише між даними програмами.

Підпис додатків є ще один кроком для захисту системи. Це надає розробникам можливість ідентифікувати себе як,автора програми та плавно оновлювати її без необхідності громіздких процесів. Програми, які не підписані розробником, не можна завантажувати в Google Play. Якщо кілька програм підписані одним сертифікатом, вони можуть вказати ключ android:sharedUserId у своєму документі, щоб вони мали однаковий UID. Програми мають можливість створювати дозволи безпеки, захищені підписом. Таким чином, програми, підписані під одним сертифікатом, під різними ідентифікаторами UID та ізольованими програмами, можуть отримати доступ до обмежених функціональних можливостей, одні одних.

1.3 Огляд вразливостей ОС Андроїд

В ОС Android було з 2009 року знайдено вразливості таких типів: DoS, Переповнення буфера, Пошкодження пам’яті, SQL Ін’єкції, XSS, Directory Traversal, Bypass something, Отримання Інформації RCE, Отримання привілей.

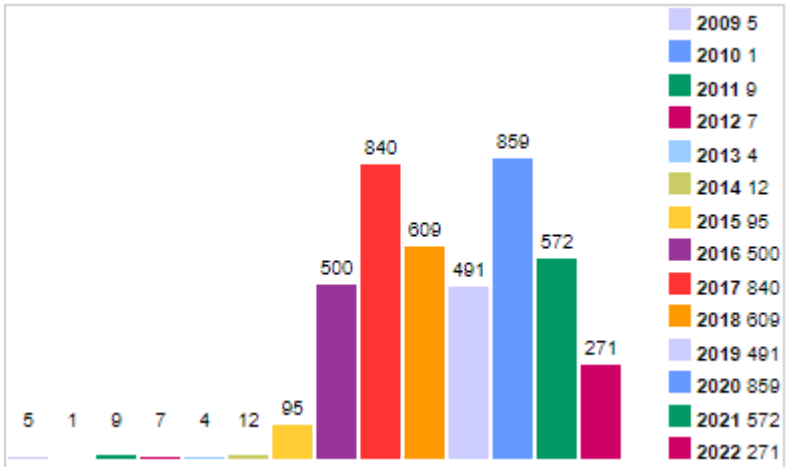

Рисунок 1.2[3] — Кількість вразливостей ОС Android за певний рік

Як бачимо з Рисунку 1.2 — найбільша кількість вразливостей була зафіксована у 2020 році - чутливими виявилися пристрої LG та Samsung Android OS 10, та операційна система Android 11.

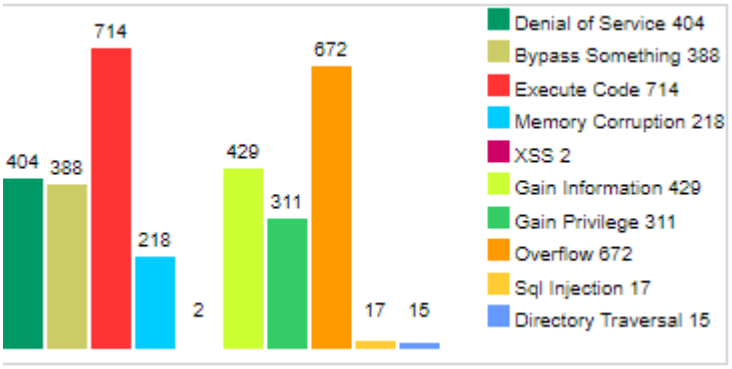

Рисунок 1.3[3] — Кількість вразливостей ОС Android за видами

З Рисунку 1.3 бачимо, що найбільша кількість вразливостей типу - Execute Code. У 2017 році було знайдено 206 вразливостей даного типу. Саме тоді відкрили один з найбільш критичних рядів вразливостей під назвою BlueBorne, що відноситься до типу Отримання інформації та RCE.

Вразливі версії Android: 4.4.4, 5.0.2, 5.1.1, 6.0, 6.0.1, 7.0, 7.1.1, 7.1.2, 8.0

Щоб здійснити дану атаку потрібно було знаходитися на відстані не більше 10 метрів від цілі на якій включений Bluetooth. Результатом було виконання

довільного коду з привілеями ядра системи та витік даних. Компанія Google 9 вересня 2017 року випустила оновлення, яке усуває даний пул вразливостей. Також пристрої з режимом Bluetooth Low Energy були не вразливі до BlueBorne.

1.4 Remote Code Execution атаки та їх типи

Remote Code Execution атака - це виконання коду, що описує форму кібератаки за якої зловмисник може отримати повний контроль над пристроєм. Атака віддаленого виконання коду може призвести до повномасштабної атаки, яка скомпрометує всю веб-програму та веб-сервер.

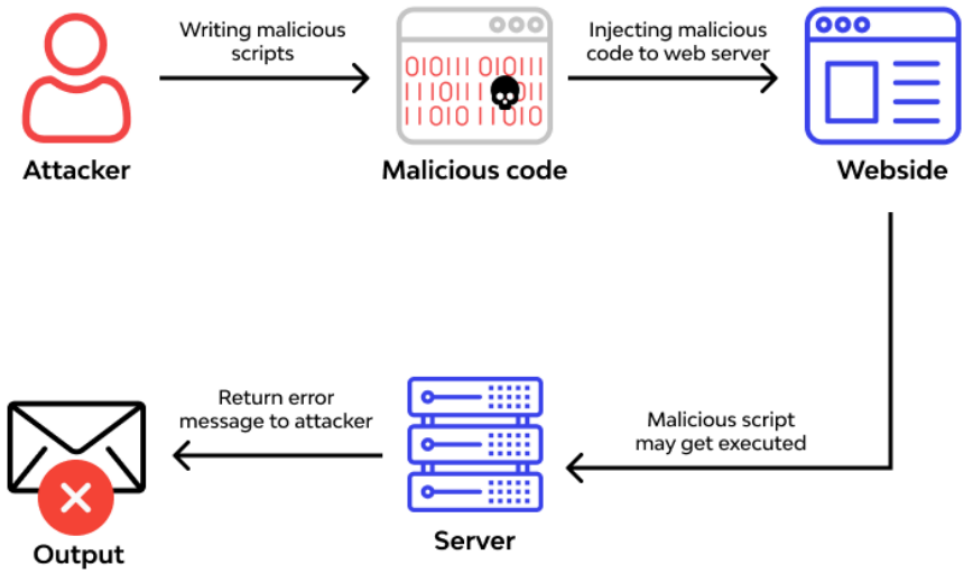

Рисунок 1.4[4] —Схема Remote Code Execution атаки

Метою проведення RCE атак є отримання неавторизованого доступу до системи. Мотивація даних атак залежить від специфіки бізнесу його клієнтів та даних, якими він оперує. Розрізняємо такі цілі RCE атак:

- Проникнення та контроль. Використання RCE атаки для отримання присутності у мережі жертви, приклад - корпоративне шпигунство. Це може бути одним із кроків ширшої атаки, даний спосіб забезпечує постійний моніторинг операцій системи.
- Викрадення даних. RCE атаки можуть надавати доступ до встановлення додаткового коду, що призведе до передачі даних. Отримання контролю над обчислювальними ресурсами жертви за допомогою RCE, що дає змогу після цього завантажити велику кількість даних.
- Порушення процесу. За допомогою RCE відбувається проникнення до системи з метою шкоди її процесам, наприклад - переповнення буфера. В результаті чого йде нанесення ударів задля порушення роботи системи, що створює затримку обробки інформації або взагалі призупиняє її, що може понести фінансові втрати для жертви.

Атаки даного типу залежать від конкретної вразливості, де зловмисник бажає отримати несанкціонований доступ до ваших систем. В даному описі надано приклади вразливостей за 2022 рік, які посортовано за трьома типами форматів атак.

1.4.1 Атаки десеріалізації

Серіалізація даних – це перетворення складних структур даних на більш плоску структуру даних, яку можна надсилати і простому послідовному потоці. Відновлення даного потоку - це десеріалізація, яка може призвести до

випадкового виконання двійкового коду. Зловмисники мають на меті змінити серіалізовані дані, вставляючи код у змінений об’єкт даних.

Приклади:

- CVE-2017-0871[5]: Уразливість із підвищенням привілеїв у платформі Android. Продукт: Android. Версії: 8.0. Android ID A-65281159.
  Оскільки код серіалізації в ParcelableException дійсно забезпечує перевірку типу для класу, записаного в Parcel, зловмиснику все одно потрібно змінити Parcel безпосередньо, щоб використати цю вразливість. Викликаючи конструктор FileOutputStream, щоб присікати довільні файли як користувач системи.
  Вихідний код було оновлено з перевіркою:
  final Class<?> clazz = Class.forName(name, true, Parcelable.class.getClassLoader());
  if (Throwable.class.isAssignableFrom(clazz)) {
  return (Throwable) clazz.getConstructor(String.class).newInstance(msg);
  }
- CVE-2021-39794[5]. У broadcastPortInfo AdbService.java є можливий спосіб для програм запускати код з оболонки користувача, якщо ввімкнено бездротове налагодження через відсутність перевірки дозволу. Це може призвести до локального підвищення привілеїв без додаткових привілеїв виконання.
  Версії Android: Android-11 Android-12 Android-12L. Ідентифікатор: A-205836329
  Для виправлення вразливості, щоб обмежити трансляції AdbManager, дадали функцію перевірки MANAGE_DEBUGGING.
  Приклад з файлу AdbManager.java

```
/**
 * @return true if the device supports secure ADB over Wi-Fi.
 * @hide
 */
@SystemApi
@RequiresPermission(android.Manifest.permission.MANAGE_DEBUGGING)
public boolean isAdbWifiSupported() {
    try {
        return mService.isAdbWifiSupported();
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

Рисунок 1.5 — Демонстрація усунення вразливості CVE-2021-39794

- CVE-2022-20005[5]. У validateApkInstallLocked пакета PackageInstallerSession.java, що є спосіб примусово встановити невідповідність між запущеним кодом і проаналізованим файлом .apk. Це може призвести до локального підвищення привілеїв із необхідними привілеями на виконання користувача.

  Версії Android: Android-11 Android-12 Android-12L. Ідентифікатор: A-219044664

  Виправлення - перевірка перезапуску програми, при оновленні base.apk.

```
// Installing base.apk. Make sure the app is restarted.
params.setDontKillApp(false);
```

Рисунок 1.6 — Демонстрація усунення вразливості CVE-2022-20005

- CVE-2021-39628[5]. Через логічну помилку у файлі - StatusBar.java можливе розкриття вмісту сповіщень на заблокованому екрані. Що може призвести до розкриття локальної інформації без додаткових привілеїв на виконання.

Версія Андроїд - 10, 11. ID: A-189575031

Виправлення: Застосування примусового змінення стану рядка під час скасованого вимкнення екрану. При вимкненні екрану показується інтерфейс користувача AOD без повного перемикання в стан KEYGUARD. При скасуванні вимкнення екрану, компоненти повертаються до стану SHADE. Отже в такому разі всі компоненти скидаються до стану SHADE.

- CVE-2021-39689[5]. У кількох функціях odsign_main.cpp існує можливий спосіб системної атаки через логічну помилку в коді. Це може призвести до локального підвищення привілеїв із необхідними привілеями на виконання системи.

  Версії Android: Android-12 Ідентифікатор Android: A-206090748

  Для усунення вразливості було додано функцію перевірки артефактів перед викликом odrefresh --compile. Odrefresh підтримує часткову компіляцію, лише повторно генеруючи файли які потрібні. Потрібно перевірити вже існуючі артефакти перед викликом odrefresh, тому що інакше не дізнаємося чи були файли, які не знаходяться в fs-verity, згенеровані odrefresh,або зловмисником, якому вдалося зламати файлову систему на попередній цикл завантаження.

```
art::odrefresh::ExitCode checkArtifacts() {
    const char* const argv[] = {kOdrefreshPath, "--check"};
    const int exit_code =
        logwrap_fork_execvp(arraysize(argv), argv, nullptr, false, LOG_ALOG, false, nullptr);
    return static_cast<art::odrefresh::ExitCode>(exit_code);
}
```

Рисунок 1.7 — Демонстрація усунення вразливості CVE-2021-39689

- CVE-2021-39695[5]. У функції через логічну помилку в коді createOrUpdate файлу з представленням системних дозволів BasePermission.java можливий

обхід дозволу. Це може призвести до локального підвищення привілеїв із необхідними привілеями на виконання користувача.

Версії Android: Android-11. Ідентифікатор: A-209607944

Для усунення даної вразливості було виправлено невідповідність рівня захисту BasePermission.perm.protection-Level і BasePermission.getProtectionLevel() в межах одного об’єкта.

```
if (bp.perm != null && Objects.equals(bp.perm.getPackageName(), p.getPackageName())
        && Objects.equals(bp.perm.getName(), p.getName())) {
    bp.perm.setFlags(bp.perm.getFlags() & ~PermissionInfo.FLAG_INSTALLED);
    bp.perm = p.setFlags(p.getFlags() | PermissionInfo.FLAG_INSTALLED);
    bp.protectionLevel = p.getProtectionLevel();
}
```

Рисунок 1.8 — Демонстрація усунення вразливості CVE-2021-39695

1.4.2. Атака переповнення буфера

В даній атаці використовується вразливість, яка дозволяє зловмиснику перезаписати дані, що зберігаються у пам’яті.Програми регулярно виділяють шматки пам’яті фіксованого розміру для зберігання даних, включаючи дані від користувача. Якщо це виділення пам’яті виконується неправильно, зловмисник може розробити вхід, який записує за межі виділеного буфера. Оскільки виконуваний код також зберігається в пам’яті, надані користувачем дані, записані в потрібному місці, можуть виконуватись.

Приклади:

- CVE-2021-39667[5]. У файлі ih264d_parse_decode_slice їx264d_parse_slice.c можливий запис поза межами через переповнення буфера купи. Це може призвести до віддаленого розкриття інформації без додаткових привілеїв на виконання.

Версії Android: 10, 11, 12. Ідентифікатор Android: A-205702093

Виправлення: Приріст фрагментів тепер виконується після завершення аналізу заголовка.Раніше це було зроблено перед перевіркою параметрів, пов’язаних із mmco, та розрахунком poc. Якщо на цьому етапі були виявлені помилки, збільшення фрагментів було неправильним.

- CVE-2021-39793[5]. У функції через логічну помилку в коді kbase_jd_user_buf_pin_pages файлу mali_kbase_mem.c можливий запис поза межами купи. Що може призвести до локального підвищення привілеїв без додаткових привілеїв виконання.

  Версії Android: kernelAndroid. Ідентифікатор Android:A-210470189
- CVE-2021-39671[5]. У коді, згенерованому за допомогою aidl_const_expressions.cpp, є можливе читання поза межами купи через неініціалізовані дані. Це може призвести до розкриття інформації без додаткових привілеїв на виконання.

  Версія Android: 12. Ідентифікатор Android: A-206718630

  Для вирішення вразливості було додано автоматичне значення за замовчуванням для поля типу char '\0', якщо нічого не вказано до вищевказаного файлу.
- CVE-2021-3968[5]. У фунції delete_protocol файлу main.c можливе виконання довільного коду, через присутність методу звільнення пам’яті. Використання раніше звільненої пам’яті може мати будь-яку кількість несприятливих наслідків – від пошкодження дійсних даних до виконання довільного коду, залежно від екземпляра та часу виникнення дефекту. Різні вказівники функцій можуть бути розкидані в даних купи. Якщо один з цих вказівників функцій буде перезаписано адресою на дійсний шелл-код, можна досягти виконання довільного коду.

  Версії Android: kernelAndroid. Ідентифікатор: A-200251074

- CVE-2021-0959[5]. Наявна логічна помилка у коді у файлі - jit_memory_region.cc, що дозволяє створити обхід обмежень пам’яті. Завдяки цьому можна отримати локальне підвищення необхідних привілеїв. Версія Андроїд - 12. ID: A-200284993
- CVE-2022-26098, CVE-2022-27568, CVE-2022-27569, CVE-2022-27571, CVE-2022-27570, CVE-2022-27572[5]. Уразливісті переповнення буфера у функціях: sheifd_create, parser_iloc, parser_infe, parser_single_iref, sheifd_get_info_image, parser_ipma бібліотеки libsimba.
  Версії Android: 10, 11, 12.
  Дану вразливість усунено за допомогою надання належної перевірки даних.

1.4.3 Атака невідповідності типів

Вид атаки реалізується за умови, що у фрагменті коду відсутні заходи щодо перевірки цілісності об’єкта. За такої умови отримуємо ризик виникнення плутанини типів, що дозволить зловмиснику проникнути в код та виконувати довільні команди, створюючи невідповідність типів об’єктів.

Приклади:

- CVE-2021-39718, CVE-2021-39731[5]. У ProtocolStkProactiveCommandAdapter::Init файлу protocolstkadapter.cpp можливий запис поза межами через неправильну перевірку меж. Це може призвести до локального підвищення привілеїв із необхідними привілеями на виконання системи.
  Версії Android: kernelAndroid. Ідентифікатор Android: A-205035540, A-205036834

- CVE-2021-39726[5]. У функції cd_ParseMsg із cd_codec.c можливе читання за межами через неправильну перевірку меж. Це може призвести до віддаленого розкриття інформації без додаткових привілеїв на виконання.

  Версії Android: kernelAndroid. Ідентифікатор Android: A-181782896
- CVE-2021-39798[5]. У Bitmap_createFromParcel файлу Bitmap.cpp можливе виконання довільного коду через відсутність перевірки меж. Це може призвести до локального підвищення привілеїв із необхідними привілеями на виконання користувача. Для експлуатації не потрібна взаємодія з користувачем.

  Версії Android: Android-12 Android-12L Ідентифікатор Android: A-213169612

  Виправлення невідповідності в описі об’єкта Bitmap_createFromParcel

```
static jobject Bitmap_createFromParcel(JNIEnv* env, jobject, jobject parcel) {
#ifdef __ANDROID__ // Layoutlib does not support parcel
    if (parcel == NULL) {
        jniThrowNullPointerException(env, "parcel cannot be null");
        return NULL;
    }
```

Рисунок 1.8 — Демонстрація усунення вразливості CVE-2021-39798

- CVE-2022-25818,CVE-2022-26092[5]. Неправильна перевірка меж у стеку UWB та у бібліотеці Quram Agif.

  Версії Android: 10, 11, 12.

  Усунення вразливостей відбулося за допомогою додавання належної перевірки логіки, що запобігає довільному запису у пам’ять.
- CVE-2021-39684[5]. У функції target_init gs101/abl/target/slider/target.c можливе виділення пам'яті RWX через логічну помилку в коді. Це може призвести до локального підвищення привілеїв без додаткових привілеїв виконання.

  Версії Андроїд: kernelAndroid. ID: A-203250788R

- CVE-2021-39804[5]. Під час повторної ініціації HeifDecoderImpl.cpp можливий збій через відсутню перевірку на нуль. Це може призвести до віддаленої постійної відмови в обслуговуванні в засобі вибору файлів без додаткових привілеїв виконання.
  Версії Android: Android-12 Android-12L Ідентифікатор Android: A-215002587
  Виправлення вразливості - додання перевірки на нуль HEIF framecount.

```
bool HeifDecoderImpl::getSequenceInfo(
        HeifFrameInfo* frameInfo, size_t *frameCount) {
    ALOGV("%s", __FUNCTION__);
    if (!mHasVideo) {
        return false;
    }
    if (frameInfo != nullptr) {
        *frameInfo = mSequenceInfo;
    }
    if (frameCount != nullptr) {
        *frameCount = mSequenceLength;
    }
    return true;
}
```

Рисунок 1.9 — Демонстрація усунення вразливості CVE-2021-39804

- CVE-2021-39693[5]. У функції через логічну помилку в коді функції onUidStateChanged файлу для керування правами для рідних програм Android AppOpsService.java існує можливий спосіб отримати доступ до місцезнаходження без видимого індикатора. Це може призвести до локального підвищення привілеїв без додаткових привілеїв виконання.
  Версії Android: Android-12. Ідентифікатор: A-208662370
  Дану вразливість усунули за допомогою додавання події отримання нового масиву InProgress під час функцій додавання/видалення. Інакше виклик функцій finish(), а потім start() може перерозподілити файл масиву, а посилання вказуватиме на старе розташування масиву.

```
/** Pool for {@link InProgressStartOpEventPool} to avoid to constantly reallocate new objects */
@GuardedBy("this")
private final InProgressStartOpEventPool mInProgressStartOpEventPool =
        new InProgressStartOpEventPool();
```

Рисунок 1.10 — Демонстрація усунення вразливості CVE-2021-39693.

- CVE-2021-39797[5]. У декількох функціях класу для отримання списку активностей, які можна запустити для поточного користувача, LauncherApps.java можливе підвищення привілеїв через логічну помилку в коді. Це може призвести до локального підвищення привілеїв без додаткових привілеїв виконання.

  Версії Android: Android-12 Android-12L Ідентифікатор: A-209607104

  Щоб усунути дану вразливість було внесено безпекові зміни в для API, пов’язаних із PendingIntent, у LauncherApps. Дозвіл довільних опцій активності під час створення PendingIntent є джерелом вразливості безпеки. Цей CL видаляє activityOptions із сайту виклику.
- CVE-2022-20113[5]. У функції mPreference файлу DefaultUsbConfigurationPreferenceController.java є можливий спосіб увімкнути режим передачі файлів через логічну помилку в коді. Це може призвести до локального підвищення привілеїв без додаткових привілеїв виконання.

  Версії Android: Android-12 Android-12L Ідентифікатор: A-205996517

  Для виправлення даної вразливості потрібно було змінити конфігурацію USB у вище вказаному файлі за замовчуванням на RestrictedPreference.

Висновки до розділу 1

В даному розділі було оглянуто архітектуру та модель безпеки ОС Android, яка має технології для захисту від необдуманих дій користувача та завантажуваних додатків. Дано загальний опис вразливостей ОС Android та класифікацію RCE атак, які були відкриті у поточному році.

Отож, за 2022 рік було знайдено 28 RCE вразливостей, 39% виникає через переповнення буфера та через невідповідність типів, 22% припадає на десереалізацію. Це означає, що в ОС Android існують проблеми із захистом пам’яті. Відповідно у наступному розділі розберемо існуючі методи захисту та їх вразливі місця. 17% вразливостей було знайдено в kernelAndroid, 80% яких пов’язані пам’яттю, інші 20% через логічну помилку у коді. Варто додати що всі вразливості пов’язані з пам’яттю знайдені у файлах написаних на C/C++ це доводить, що потрібно приділяти більше уваги до статичних та динамічних методів аналізу коду при розробці. Дані методи оглянемо у наступному розділі. Для виправлення 50% помилок з пам’яттю, використали метод додавання перевірки меж для масивів. У ряді вразливостей, що виникали через логічні помилки у коді 75% припадає на файли Java, 50% яких відповідають за керування певними дозволами. Для 21% вразливостей, ще не опублікували оновлення з виправленнями, що може призвести до їх експлуатації.

Таким чином постає завдання аналізу існуючої системи методів захисту від RCE атак для ОС Android.

2 ОПИС ТА АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ПІДВИЩЕННЯ РІВНЯ ЗАХИЩЕНОСТІ ОС ANDROID

2.1 Методи захисту від десеріалізації

Для захисту від даної атаки не існує єдиного рішення. При запобіганню десереалізації відповідно до логіки певної програми, даної загрози усунути взагалі не вийде, через те що інші компоненти у стеку даної програми будуть використовувати процес десереалізації. Розглянемо та проаналізуємо методи захисту від десереалізації для ОС Android.

1) Самозахист додатків під час виконання (RASP)[6] — це технологія, яка виникає під час запуску програми. Вона перехоплює виклики з програм до систем, щоб переконатися, що вони безпечні, і перевіряє запити безпосередньо всередині програми. Програмне забезпечення RASP вбудовується або інтегрується з середовищем виконання програми, щоб виконувати свої функції щодо захисту програми. Ще однією перевагою даної технології є те що вона підтримує хмарні рішення. У джаві це імплементовано за допомогою включення до стандартних бібліотек рівня RASP, який прослуховує виклики до допоміжних бібліотек і застосовує правила, коли виклики перехоплюються. Водночас дана технологія може уповільнити час виконання програми до 50% і більше.
2) Мовно-агностичний метод — це використання автентифікаваного підпису для сереалізованих даних та чистих форматів даних, таких як: JSON або XML, для уникнення нативних форматів десеріалізації. Водночас можливість того що програма все ще зробить помилки безпеки після аналізу чистого об’єкта даних існує.
3) Використання безпекових бібліотек при написанні коду

Для десереалізації у Java можна використовувати бібліотеку — SerialKiller, що перевіряє класи Java під час розділення імен і дозволяє комбінувати чорний/білий список для захисту програми.

2.2 Методи захисту від атак переповнення буфера

1) Захист, який пропонує ОС
   - ASLR[7], рандомізація місць пам’яті структур. Це включає базу виконуваного файлу та позиції стека, купи та бібліотек. ASLR був розроблений для захисту, де обхідному шляху до невиконуваних стеків з’єднуються в ланцюг на основі зміщень їх адрес у пам’яті, існуючі фрагменти коду. Такий підхід ускладнює для зловмисника надійний перехід до певної функції в пам’яті. Цю техніку було застосовано в Android 4.0. Дана технологія є корисною при застосуванні бібліотек із C та є вразливою до побайтової атаки, так як рандомізація відбувається не у всіх областях пам’яті, або відбувається лише при завантаженні системи. Водночас існуючі системні конструкції в Android можуть викликати конфлікт з ASLR, що призведе до його обходу. Наприклад модель створення процесу Zygote, що зроблена для підвищення продуктивності програми, може значно знизити ефективність ASLR. У Zygote, код програми завжди завантажується в одному і тому ж місці пам’яті під час різних запусків, навіть якщо ASLR присутній; всі запущені програми успадковують зазвичай використовувані бібліотеки від процесу Zygote (включаючи бібліотеку libc) і, таким чином, спільно використовують ті самі відображення віртуальної пам’яті цих бібліотек, які представляють загалом 27 МБ виконуваного коду в пам’яті, що дозволяє виконати атаку поворотно орієнтовного програмування.

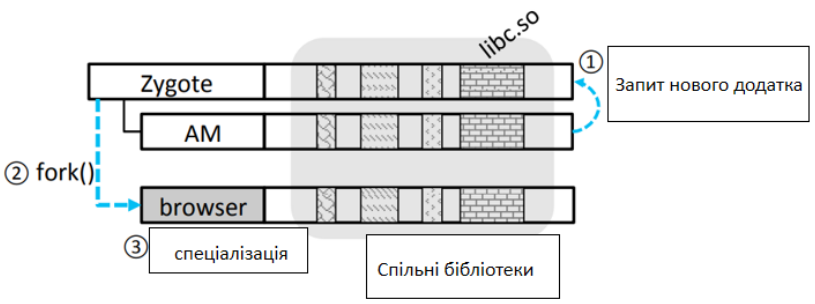

Рисунок 2.1 — Модель створення процесу Zygote

На Рисунку 2.1 бачимо представленням макета пам’яті під час запуску програми для браузера. AM представляє менеджер діяльності. Сіра область відображає знімок макетів адресного простору Android для запущених програм, безпосередньо успадкованих від процесу Zygote; кожен шаблон представляє спільну бібліотеку, таку як libc.so або libssl.so, яка розташована ідентично серед усіх запущених програм.

- DEP — технологія що дозволяє програмі виконувати код тільки в області пам'яті, позначеної як виконувана. Якщо шкідливий код попадає в область даних. яка позначена як невиконана, він працювати не буде. Сторінка віртуальної пам'яті, дані в якій не призначені для виконання, позначається бітом NX (у термінології AMD) і XD бітом (по Intel). DEP запобігає не тільки атакам, але й помилкам операційної системи, просто завершуючи програму. Якщо не підтримується біт NX, тоді застосовується програмна реалізація DEP. Вона істотно поступається апаратною, тому що захищає лише окремі системні файли операційної системи, але при цьому працює на будь-яких процесорах. DEP включений в Android, починаючи з версії 4.1 Jelly

Bean. Проте через відсутність контролю меж локальних буферів можна затерти адресу повернення з функції для переміщення сюди покажчика на шелл-код або, скориставшись переповненням купи, перезаписати будь-яку комірку, що модифікується, в адресному просторі вразливого процесу.

- Scudo[8] — це динамічний розподільник пам’яті в режимі користувача або розподільник пам’яті, розроблений, щоб бути стійким до вразливостей, пов’язаних з купою, зберігаючи при цьому продуктивність. Він забезпечує стандартні примітиви розподілу та звільнення C (такі як malloc і free), а також примітиви C++ (наприклад, new і delete). Починаючи з випуску Android 11, scudo використовується для всього рідного коду.

2) Захист на рівні прав доступу для користувача

Принцип найменших привілеїв(POLP[9]). Концепція за якої користувачам і додаткам слід надавати лише ті дозволи, які їм необхідні для виконання своїх завдань або виконання завдань, які їм призначені. Дотримання підходу POLP зменшує ймовірність атаки переповнення буфера. Хоча POLP допомагає мінімізувати ризик доступу неавторизованого користувача/процесу до конфіденційних даних, основним недоліком є те, що мінімальні дозволи повинні відповідати ролям та обов’язкам користувача, що може бути складним при великій кількості користувачів. Наприклад, користувачі можуть бути не в змозі виконати певне обов’язкове завдання, якщо вони не мають відповідних повноважень. Імплементація даної технології в ОС виглядає так:

- перевірка відсутності збоїв привілеїв для всіх існуючих програм та процесів;
- запуск програм з найменшими привілеями та додавання привілеїв відповідно до доступу, необхідного для виконання;

- відстеження окремих дій, що здійснюються за допомогою одноразових облікових даних та процесів, щоб уникнути потенційної шкоди.

Перевагами даного методу є запобігання поширенню шкідливих програм. З допомогою обмежень POLP, ШПЗ не зможе використовувати облікові записи з вищими привілеями або облікові записи адміністратора для встановлення зловмисного програмного забезпечення або пошкодження системи.

3) Інструменти компілювання
   - ShadowCallStack (SCS)[10] — це режим інструментування LLVM, який захищає від перезапису адреси повернення, зберігаючи адресу повернення функції в окремо виділений фрагмент. Адреса повернення також зберігається в звичайному стеку для сумісності з розмотувачами, але в іншому випадку не використовується. Це гарантує, що атаки, які змінюють адресу повернення в звичайному стеку, не впливають на потік керування програмою.
   - Stack Smashing Protector (SSP) — це функція компілятора, допомагає виявити переповнення буфера стека шляхом переривання, якщо секретне значення в стеку змінено. Стек канарейки має розмір рідного слова, і якщо його вибрано випадковим чином, зловмиснику доведеться вгадати правильне значення. В Android опорне значення канарки зберігається в глобальній змінній, це означає що однакове опорне значення канарки буде використовуватися всіма програмами користувача Android, запущеними в системі. Тож будь-яка локальна програма Android знає канаркове значення будь-якої іншої програми, що дозволяє зловмиснику додати просту «троянізовану» програму до «Google Play», яка буде надсилати йому значення комірки-канарки разом з іншою корисною інформацією. Як тільки зловмисник

дізнається значення канарки, будь-яка помилка переповнення буфера, доступна будь-якій запущеній програмі Android, може бути використана для виконання шкідливого коду шляхом зміни адреси повернення стека. Також основними недоліками даної технології є те що як тільки зловмисник отримує значення канарки, він може використовувати його, якщо система не буде перезавантажена, що може тривати досить тривалий період часу. Отримане канаркове значення можна навіть використовувати проти програм, встановлених після витоку канаркового значення.

- AddressSanitizer/HWAddressSanitizer[11] — це інструментальна компонента на основі компілятора, що виконує динамічний та статичний аналіз. AddressSanitizer поєднує інструменти всіх викликів функцій, пов’язаних з пам’яттю, включаючи alloca, malloc і free, і заповнення всіх змінних і виділених областей пам’яті, що дозволяє виявляти помилки недійсного використання пам’яті, включаючи подвійне звільнення та повернення , тоді як заповнення області пам’яті виявляє читання або запис за межі. Якщо відбувається читання або запис в цю область заповнення, Address Sanitazer ловить це і виводить інформацію, яка допомагає діагностувати порушення пам’яті, включаючи стек викликів, карту тіньової пам’яті, тип порушення пам’яті, те, що було прочитано або записано, інструкцію, яка викликала порушення та вміст пам'яті. Починаючи з Android 11 даний інструмент замінили на аналог — HWAddressSanitizer, що допомогло зменшити витрати ЦП та ОЗП. Також, HWASan виявляє використання стека після повернення.За конструкцією HWASan не має обмежених червоних зон як ASan для виявлення переповнень. Відповідно до цього HWASan може виявити помилку незалежно від того, наскільки великим є переповнення або як давно пам’ять була

звільнена, що дає певну перевагу над Asan. HWASan не може бачити код Java як і ASan, але може виявляти помилки в бібліотеках JNI. На пристрої програми можна перевірити за допомогою HWASan, додавши в їх код SANITIZE_TARGET:=hwaddress у Make або -fsanitize=hwaddress у прапорах компілятора.

2.3 Методи захисту від атак невідповідності типів

1) Статичний аналіз

WARNING_PATH_SENSITIVE_USE_GSL_AT[12] — це нове правило, що було додано до статичного аналізу в Visual Studio 16.10. Воно аналізує стандартні контейнери на предмет неперевіреного доступу до елемента та чутливе до шляху, для відстеження викликів перевірки розміру буфера. Однак перевірка, може забирати досить багато ресурсів, так як час необхідних для аналізу кожної функції залежить від залежать від довжини, кількості операцій розгалуження та окремих властивостей. Воно обходить функцію та емулює кожну гілку та цикл, з якими вона зустрічається, оновлюючи внутрішній стан на основі різних припущень, зроблених у коді.

2) Технологія від Android

BoundsSanitizer (BoundSan)[13] — технологія, що додає інструменти до бінарних файлів, щоб вставляти перевірки меж навколо доступу до масиву. Ці перевірки додаються, якщо компілятор не може довести під час компіляції, що доступ буде безпечним, і якщо розмір масиву буде відомий під час виконання, щоб його можна було перевірити. Android 10 розгортає BoundSan в Bluetooth і кодеках. BoundSan надається компілятором і вмикається за замовчуванням у різних компонентах платформи. Недоліком може стати навантаження на продуктивність, яке залежать від кількості доступів до масиву, які не можуть бути безпечними.

Висновки до розділу 2

В даному розділі було розглянуто існуючі методи захисту від різних типів RCE атак. Умовно контрзаходи від атак, які нам пропонуються можна поділити на такі види:

- на основі ОС Android:
- на основі компілятора
- на основі бібліотек
- моніторинг виконання

Було виявлено конфлікт реалізації ASLR з моделлю створення процесу Zygote, що послаблює дану технологію захисту від переповнення буфера. Розглянемо варіант пом’якшення даної проблеми у наступному розділі. Також описано методи захисту на основі компілятора, один з яких перекриває вразливість у дизайні ОС Android - зберігання канаркового значення у глобальній змінній. Виявлено закономірність, що методи захисту на основі бібліотек застосовуються до файлів Java, на основі компілятора для C/C++. Водночас описані вище методи захисту від десеріалізації та атак невідповідності типів несуть навантаження на продуктивність та збільшують час роботи програм.

Найбільш актуальними методами захисту від атак RCE є аналіз коду, для уникнення логічних помилок, налагодження та компіляція, використання безпекових бібліотек, фільтрація вводу, керування пам’яттю та контроль доступу.

3. ОПИС ЗАПРОПОНОВАНИХ МЕТОДІВ ДЛЯ ЗАПОБІГАННЯ CODE EXECUTION ВРАЗЛИВОСТЕЙ

Захист пам’яті в ОС Android є досить важливим аспектом превентивних заходів від RCE атак. Так як було виявлено певний конфлікт в реалізації ASLR та методу запуску Zygote, пропонується розглянути 2 рішення - Morula та SSPFA.

3.1 Morula[14]

Це модель створення процесу, яка випадково уніфікує розташування критичних областей пам’яті в усіх запущених програмах, що пом’якшує негативні побічні ефекти на ASLR.

Розглянемо функцію налагодження в ОС Android process_wrap, що забезпечує хук для створення процесу, який викликається відразу після того як процес відділяється від Zygote і перед тим як будь-які дані програми будуть завантаженні в даний процес. Модель створення процесу Wrap змушує кожен процес, відділений від Zygote, відновлювати свої макети пам’яті за допомогою виклику exec(), що перезавантажує образ процесу та спільні бібліотеки. Це призводить до того що ASLR рандомізує пам’ять окремо для кожного процесу, що відбувається як частина виклику exec().

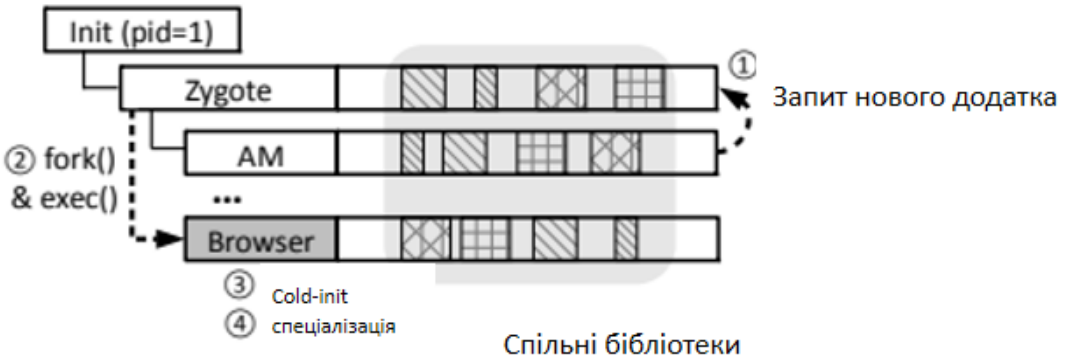

Рисунок 3.1[15] — Робочий процес Wrap моделі

1) Робочий процес даної моделі виглядає так:

   Спочатку Activity Manager надсилає запит на створення програми до процесу Zygote.
2) Zygote розгалужує новий процес, який викликає exec(), щоб завантажити головне зображення процесу програми (/system/bin/app_process), в результаті перехоплення process_wrap.
3) Далі йде ініціалізація процесу, яка не потрібна в моделі Zygote
4) Етап спеціалізації, де встановлюються атрибути процесу та потік керування переходить до точки входу цільової програми .

Тепер, коли Android дотримується типової моделі створення процесу fork-then-exec, кожен процес має унікальну рандомізовану компоновку пам’яті, де спільні бібліотеки більше не однаково розташовані в пам’яті, як показано в затіненій області на Рисуноку 3.1. Проте дана модель значно уповільнює запуск програми, так як повинна проходити повний цикл створення та ініціалізації DVM, чого уникає модель Zygote, це призводить до середнього часу запуску програми

4,34 секунди, додаючи 3,52 секунди очікування. Це доводить, модель створення процесу Zygote є критичною для продуктивності конструкцією в Android, незважаючи на побічний ефект у послабленні ASLR, яку неможливо просто видалити. Рішення безпеки, спрямовані на усунення негативних побічних ефектів, повинні мати можливість зберігати переваги її продуктивності.

Morula дозволяє процесам додатків мати індивідуальні рандомізовані макети пам’яті, як і в моделі Wrap, проте зберігає час виконання такий як Zygota, що вдається завдяки попередньому виконанню загальних та трудомістких завдань ініціалізації. Дана модель створена на основі Zygote, вона переводить роль її процесу на абстрактний шаблон, що розділяє проміжні шаблони процесів. Заздалегідь створені процеси є ініціалізованими хостами для виконання додатків. Даний процес миттєво завантажує та запускає програму без повторення трудомістких завдань ініціалізації відразу як отримує запит на запуск цієї програми. Кожен процес даної моделі повинен перезавантажувати незалежне зображення пам’яті під час етапу попередньої ініціалізації, це гарантуватиме що дана розбивка пам’яті не буде розподілена між різними процесами програми. Також процеси Morula створюються асинхронно для майбутніх подій запуску програми, вони переходять у сплячий режим, а потім переходять до наступної фази після пробудження.

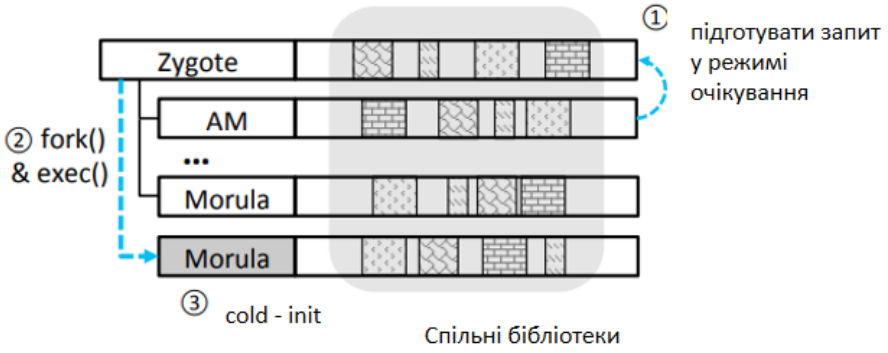

Рисунок 3.2[15] — Підготовча фаза моделі Morula

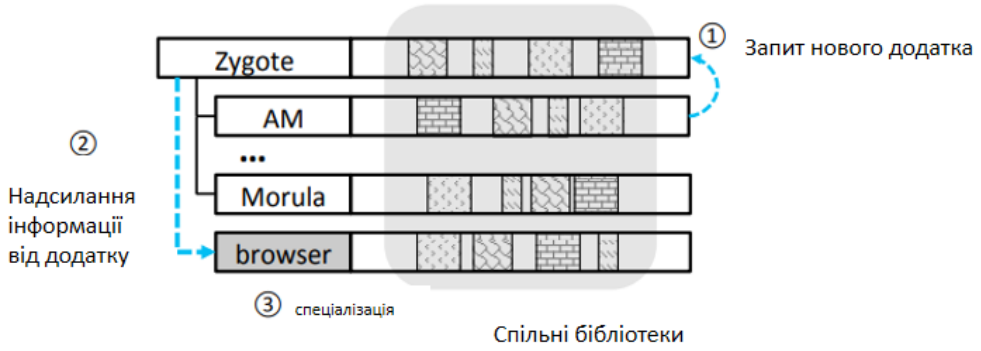

Рисунок 3.3[15] — Перехідні фаза моделі Morula

На Рисунку 3.2 показано фазу, де процес Morula готується, а потім використовується для запуску процесу програми - браузер. Підготовчий етап:

1) Ініціація Менеджером діяльності через запит підготовки до процесу Zygote, коли система неактивна.
2) Процес Zygote розгалужує дочірній процес, який негайно викликає exec(), щоб створити нове зображення пам’яті зі свіжо рандомізованим макетом.
3) Новий процес створює екземпляр DVM і завантажує всі спільні бібліотеки та загальні класи Android. Після цього створюється процес Morula, який очікує запиту на запуск програми.

На Рисунку 3.3 показана перехідна фаза:

1) Процес Morula входить у перехідну фазу за запитом Менеджера діяльності, щоб запустити нову програму. Запит надходить до процесу Zygote, тут приймається рішення в якій саме моделі запускати програму.
2) Процес Zygote пересилає запит на запуск програми з конкретною інформацією про додаток.

3) Процес починає спеціалізуватися, завантажуючи пакет програми та встановлюючи відповідний UID, GID, прапорці налагодження. Процес Morula передає потік керування програмі та переходить у процес браузера

Дизайн даної моделі є гібридом, що поєднує модель Zygote та модель Wrap. З першої вона взяла попереднє виконання трудомістких завдань пов'язаних зі створенням процесів додатків. З іншої — забезпечення оновлення макета пам'яті під час попереднього створення процесів шаблону. Тож, модель Morula створює індивідуальний рандомізований макет пам'яті в програми Android, зберігаючи при цьому оптимальний час запуску програми.

Модель Morula викликає додаткові витрати часу та продуктивності в порівнянні з моделлю Zygote. Щоб зменшити навантаження на систему модель може використовувати оптимізацію — вибіркову рандомізацію. Що дає можливість використовувати Morula лише для запуску програм що можуть бути вразливими до ASLR-експлойтів, а інші програми запускатимуться за моделлю Zygote.

3.2 SSPFA

SSPFA[16] базується на тій же інфраструктурі що й SSP, зміни відбуваються у Zygote, а саме з'являється функція оновлення еталонної канарки для дочірнього процесу, відразу після його створення. При чому значення еталонної канарки Zygote залишається незмінним.

Дана модифікація дозволяє зберегти нормальну роботу програми завдяки тому що:

- У Zygote після операції fork() дочірній процес виконує потік коду, який закінчується явним викликом exit();
- Для кожного процесу існує одна опорна канарка, яка зберігається в відокремленій області пам'яті та ініціалізується під час запуску процесу;

- Перевірка кадрової канарки з опорною канаркою виконується лише в кінці кожної функції або блоку коду, безпосередньо перед інструкцією повернення;
- З контрольною канаркою перевіряється лише значення кадрової канарки поточного стеку.

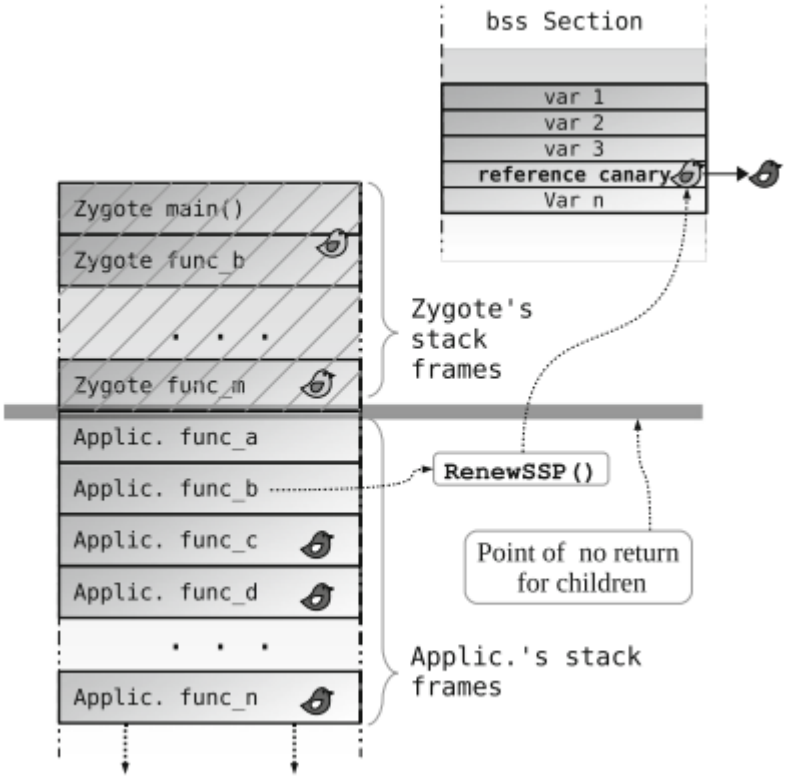

Рисунок 3.4[16] — Активний стек програми

На Рисунку 3.4 показано поділ стеку після зміни контрольної канарки. Кадри стеку, що пов’язані з програмою Zygote, мають однакове опорне значення канарки. Наступним кроком викликається нова функція renewssp() паралельно викликаючи точку неповернення. Відтепер кожна програма матиме різне еталонне значення канарки, що дозволить уникнути експлуатації даної вразливості.

Форма потоку виконання — setjmp/longjmp, використовується як механізм винятків для повернення до початкового безпечного стану. Функція longjmp() не

перевіряє цілісність стеку функцій поточних та призначення, що дозволяє її використовувати після змінення еталонного значення канарки. За умови якщо значення еталонної канарки не буде постійним протягом усього процесу його потрібно зберігати під час виклику setjmp() разом з рештою інформації про середовище.

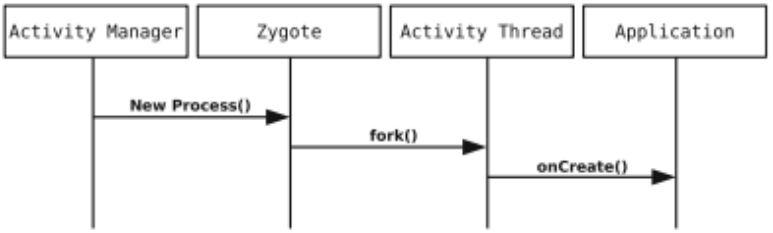

Рисунок 3.5[16] — Запуск нового процесу в ОС Android

Для запуску нового процесу як бачимо з Рисунку 3.5 в ОС Android існує три етапи:

1) створення процесу, який створює структури в пам’яті для виділення процесу
2) прив’язка програми, яка пов’язує скомпільований програмою код із структурами процесу
3) обхід потоку, який передає керування потоком до програми.

SSPFA реалізується на першому етапі, у місці де нова програма і код Zygote розділяються, саме там оновлюється опорне значення канарки. Під час даного етапу Android ActivityManager надсилає до Zygote запит на створення нового процесу через сокет підключення. Потім Zygote розгалужує новий процес і створює екземпляр об’єкта ActivityThread, який запускає конкретну програму, викликаючи функцію onCreate(). Даний метод не змінює логіку Zygote, що дозволяє використовувати його в різних версіях ОС Android. Також варто додати, що так як реалізація SSPFA спирається на вже існуючу інфраструктуру SSP, вона не потребує додаткового сховища ні глобального, ні локального кадру стека.

SSPFA вимагає певне обмеження, після зміни опорної канарки не дозволяється повертатися з попередніх функцій стека, якщо ці функції перевіряють канарку. Проте воно не впливає на роботу системи тау як за замовчуванням повернення до батьківського коду після операції fork() відбувається вкрай рідко, дочірні процеси виконують інший потік коду, який завжди закінчується викликом exit().

Висновки до розділу 3

У даному розділі запропоновано покращені методи захисту для наявного ASLR, який є у системі Android.

Описано роботу моделі Morula, яка є надбудовою над Zygote та показано, що вона може зберігати продуктивність моделі Zygote завдяки рандомізації за запитом. Зазначено, що спрощена модель Morula — Wrap, значно уповільнив запуск програм і тим самим поступається ефективнішій Morula.

Також розглянуто метод SSPFA, що є надбудовою над існуючим SSP, описано алгоритм роботи. Даний метод додає лише одну функцію до Zygote, що в подальшому не значно впливає на час роботи процесів.

Показано що дані методи не змінюють логіку Zygote, що дозволяє їх використовувати у різних версіях ОС Android та зберігають продуктивність роботи процесів.

4 РЕАЛІЗАЦІЯ

4.1 Технічний опис системи

Для впровадження даних моделей захисту було обрано ОС — Android 11, апаратне забезпечення — Samsung Galaxy A32, RAM 4 ГБ / 64 ГБ вбудованої пам'яті та для створення модифікованої ОС Android використовувалася — Ubuntu 18.04.6 LTS (Bionic Beaver) на програмному забезпеченні для віртуалізації — VMware Workstation 16.1.0.

4.1.1. Встановлення надбудови Morula

Інструмент Morula впроваджено поверх поточного менеджера процесів ОС Android. Менеджер активності розширений, щоб динамічно підтримувати пул процесів Morula та виконувати запити на запуск програми за допомогою підготовленого процесу Morula або роздвоєного процесу Zygote, залежно від стратегії вибіркової рандомізації. UID відправника завжди перевіряється, щоб переконатися, що тільки системні модулі можуть взаємодіяти з Morula, уникаючи зловживань або атак з боку шкідливих програм. В роботі використовуються модулі -m та -n — Рисунок 4.1.

```
yara@ubuntu:~/mob$ python morula_property_setup.py
Usage: morula_property_setup.py [options]

Options:
  -h, --help            show this help message and exit
  -z ZYGOTE, --zygote=ZYGOTE
                        Enable Zygote model
  -w WRAP, --wrap=WRAP  Enable Wrap model
  -m MORULA, --morula=MORULA
                        Enable Morula model
  -d DEMAND, --demand=DEMAND
                        (optimization) on-demand preloading
  -n NATIVE, --native=NATIVE
                        (optimization) randomize only native apps
```

Рисунок 4.1 — Опції інструменту Morula

4.1.2. Інтеграція SSPFA

Для застосування даного методу було внесено зміни до файлу Zygote.cpp — Рисунок 4.2. Надано доступ до функції renew_ssp() у функції у функції forkAndSpecialize Common(), яка повторно рандомізує еталонну канарку. Функція forkAndSpecializeCommon() викликається з функцій forkSystemServer() і forkAndSpecialize(). Ці функції використовуються для запуску нових дочірніх служб для системних служб і програм загального призначення.

```
static pid_t forkAndSpecializeCommon
-------
{
-------
 dvmDumpLoaderStats("zygote");
 pid = fork();

 if (pid == 0) {
  int err;
 /* The child process */

 #ifdef HAVE_ANDROID_OS
     renew_ssp();
 extern int
 gMallocLeakZygoteChild;
 gMallocLeakZygoteChild = 1;

------
 return pid;
}
```

Рисунок 4.2 — Зміни у файлі Zygote.cpp

Завдяки такому включенню функції, будь-яка рідна програма, скомпільована з новою бібліотекою, зможе викликати її за бажанням. Ця функція не отримує жодного параметра від користувача, тому неможливо скинути або встановити попередньовідоме значення значення еталонної канарки.

4.2 Аналіз ефективності застосування методів Morula та SSPFA

4.2.1 Оцінка Morula

Для оцінки ефективності впливу інструменту Morula на ASLR, проводимо дослідження випадковості розташування пам’яті всієї системи Android. Спершу визначаємо міру випадковості макета пам’яті та використовуємо її для перевірки достатньої випадковості у системі макета адресного простору. Якщо перевірка справджується, це означатиме, що зловмисник не зможе вгадати цільові адреси.

Вимір випадковості макета адресного простору кожної спільної бібліотеки відбувається за допомогою ентропії, яка фіксує невизначеність даної випадкової величини. Можливі адреси розглядаються як випадкова величина.

Xm — дискретна випадкова величина з базовими адресами {x1, x2, ..., xn} для спільної бібліотеки m, а p(xi) — функція ймовірності (pmf). Тоді H(Xm), нормалізована ентропія макета адресного простору спільної бібліотеки m, визначається як:

$$H(\mathbf{X}_m) = -\sum_{i=1}^{n} p(x_i)\frac{\ln p(x_i)}{\ln n}, \tag{4.1}$$

і $0 \leq H(Xm) \leq 1$ завдяки нормалізаційному коефіцієнту ln n. H(Xm) стає нульовим, коли спільна бібліотека m зіставляється з однією адресою для всіх різних програм, і стає одиницею, коли вона зіставляється з усіма різними та унікальними адресами.

На прикладі даних бібліотек, покажемо роботу —libc.so завантажується чотирма програмами Android за однією адресою 0x1000. Тоді вихід pmf буде p(0x1000) = 1, а ентропія макета адресного простору libc.so обчислюється як H(Xlibc.so) = 0. Якщо libssl.so завантажується чотирма програмами Android на

різні адреси {0x1000, 0x2000, 0x3000, 0x4000}, то вихідні дані pmf завжди будуть 0,25, оскільки кожна адреса розподілена рівномірно, а H(Xlibssl.so) = 1.

На основі випадковості для кожної спільної бібліотеки вище, випадковість макета адресного простору всієї системи Android на пристрої D визначається як:

$$R(D) = \frac{\sum_{m \in M} H(\mathbf{X}_m)}{|M|} \tag{4.2}$$

M — набір спільних бібліотек, запущених на пристрої D і |M| є розміром M. Таким чином, R(D) показує усереднену випадковість розташування адресного простору для всіх спільних бібліотек, що працюють на пристрої D.

Виміряно R(D)(Таблиця 4.1) на обраному пристрої Android після завершення завантаження. Для процедури завантаження були застосовані різні моделі створення процесу.

Таблиця 4.1 — Значення R(D) на тестовому пристрої Android

Mode	R(D)
Zygote	0.126
Morula	0.991

З таблиці бачимо, що система без надбудови має ентропію 0,126, це означає що поточні спільні бібліотеки серед додатків Android здебільшого поділяють свої макети адресного простору. Ентропія для Zygote не дорівнює нулю, оскільки кілька спільних бібліотек завантажуються програмами незалежно, що призводить до різних базових адрес для таких бібліотек. Morula має ентропію — 0,991, це

означає те, що система Android має різні макети адресного простору для програм, і, отже, зловмисникам важко правильно вгадати макети адресного простору.

4.2.2. Оцінка SSPFA

Для оцінки SSPFA, було використано аналітичний метод аналізу — порівняння роботи поточної та модифікованої системи. Перевірка відбувалася за допомогою читання значень еталонних канарок з пам’яті процесу за допомогою /proc/ < pid > /mem в обох версіях Android.

З Таблиці 4.2 бачимо, що всі дочірні елементи Zygote мають однакове опорне значення у стандартній системі, але різні значення при використанні модифікації SSPFA. Що і потрібно було довести.

Таблиця 4.2 — Значення еталонних канарок на тестовому пристрої Android при різних моделях створення процесу

Додаток	Процес	SSP канарка	SSPFA канарка
Zygote	zygote64	0xed82fe42a74f868e	0x562a63e6d3281ad1
System	system server	0xed82fe42a74f868e	0xe5a7ef6527cdb8d5
Phone	com.android.phone	0xed82fe42a74f868e	0x61c473b476f5c3a6
Calendar	com.android.calendar	0xed82fe42a74f868e	0x7252ef0cd718408d
Chrome	com.android.chrome	0xed82fe42a74f868e	0x73c86726ed763f77

4.2.3. Оцінка захисту від brute force атаки на ASLR

Розглянемо модель атаки на ASLR і виміряємо необхідні зусилля зловмисника для обходу ASLR. У цій моделі передбачається, що зловмисник вже отримав частину адресної інформації: адресу x для певної програми ai. Далі зловмисник намагається визначити семантично еквівалентні адреси в інших програмах. Дві адреси в окремих програмах є семантично еквівалентними адресами, якщо вони вказують на одну бібліотеку з однаковим зміщенням. Також передбачається, що зловмисник має попередні знання про розташування пам'яті, оскільки семантично еквівалентні адреси знаходяться в подібних місцях для двох програм, та доступ до оракул рішення, щоб перевірити, чи є дана адреса семантично еквівалентні в іншій програмі. На практиці прикладом того, як запитувати оракул, є використання цільової програми з вгаданою адресою, а потім прийняття рішення на основі результату: прийняти, якщо експлуатація була успішною, і відхилити, якщо експлуатація завершилася невдало.

```
// In : an address ptr leaked by an attacker
// Out: a semantically equivalent address in
//      the other target application

// walk away from ptr, by PAGE_SIZE step
for (offset = 0;; offset += PAGE_SIZE) {
  for (target in {ptr-offset, ptr+offset}) {
    // found semantically equivalent address
    if (ORACLE(target) == ACCEPT)
      return target;
  }
}
```

Рисунок 4.3 — Алгоритм атаки[14]

Враховуючи витоку адреси x, зловмисник намагається знайти семантично еквівалентну адресу для іншої цільової програми. Зловмисник неодноразово запитує оракул рішення, починаючи з витоку адреси x і переходячи вгору/вниз, додаючи/віднімаючи поточну адресу зі зміщенням сторінки. Це пояснюється тим, що базові адреси в бібліотеці вирівняні за межами сторінки, а семантично

еквівалентні адреси розташовані в подібних областях, згідно з попередніми знаннями зловмисника. Таким чином, оскільки кожен доступ до оракула передбачає додаткові витрати, необхідні зусилля зловмисника для обходу ASLR безпосередньо пов’язані з кількістю доступів до Oracle. Розглянемо кількість доступів Oracle як кількість випробувань та порахуємо кількість даних спроб використовуючи address space layout одразу після завантаження пристрою.

Таблиця 4.3 — Значення кількості доступів до Oracle

Zygote	Morula	SSPFA
1	18,350	19,363

Отже, як бачимо з Таблиці 4.3 зловмиснику потрібно шукати понад 18 000 сторінок пам’яті або завершити роботу програми понад 18 000 разів, щоб правильно вгадати адресу.

4.3 Аналіз впливу методів на продуктивність ОС Android

Щоб проаналізувати вплив на продуктивність, було оцінено кожну модель створення процесу окремо. Для цього застосовували вимір часу завантаження — вимірюється шляхом запису часових позначок журналів, які сигналізують про завершення завантаження пристрою та розмір використовуваної пам’яті — читання даних з /proc/meminfo, який показує інформацію про системну пам'ять, яку підтримує ядро Linux.

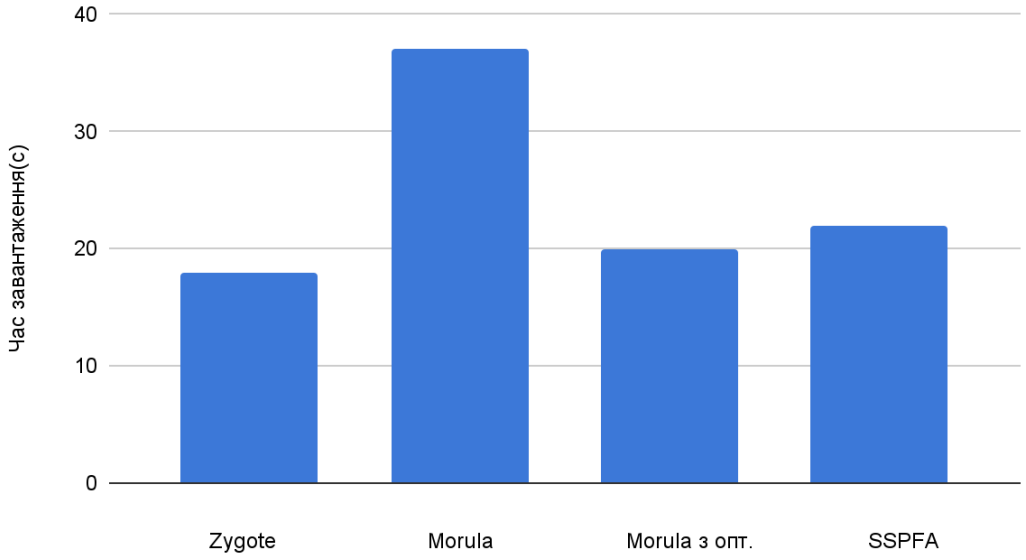

Рисунок 4.4 — Час завантаження пристрою при різних моделях

Дані вимірювання(Рисунок 4.3) показують, що Morula має більші накладні витрати, ніж Zygote до оптимізації. Це відбувається через те що багато системних програм за замовчуванням запускаються одночасно під час процесу завантаження для підтримки основних функцій Android, для прикладу com.android.phone виконується для підтримки функцій телефонних дзвінків, а com.android.nfc запускається, для увімкнення функції NFC. У тестовому середовищі оцінки під час процесу завантаження запускається 26 програм.

Час завантаження пристрою повільніший ніж у Zygote під час завантаження під Morula на 210% (10с), Morula з оптимізацією — 111%(2с), SSPFA — 122% (4с). Такі витрати Morula пов’язані з тим що, інструмент потребує більше часу для виконання кожної програми для ініціалізації DVM та переміщення адрес, які не потрібні в моделі Zygote. При SSPFA утримується час на рандомізацію канарки для кожного процесу, що уповільнює процес завантаження. Завдяки вибірковій рандомізації у варіанті Morula з оптимізацією, лише 5 вимагали даного модуля, що і скоротило час завантаження пристрою. Даний метод має найменшу різницю з рідною моделлю Android.

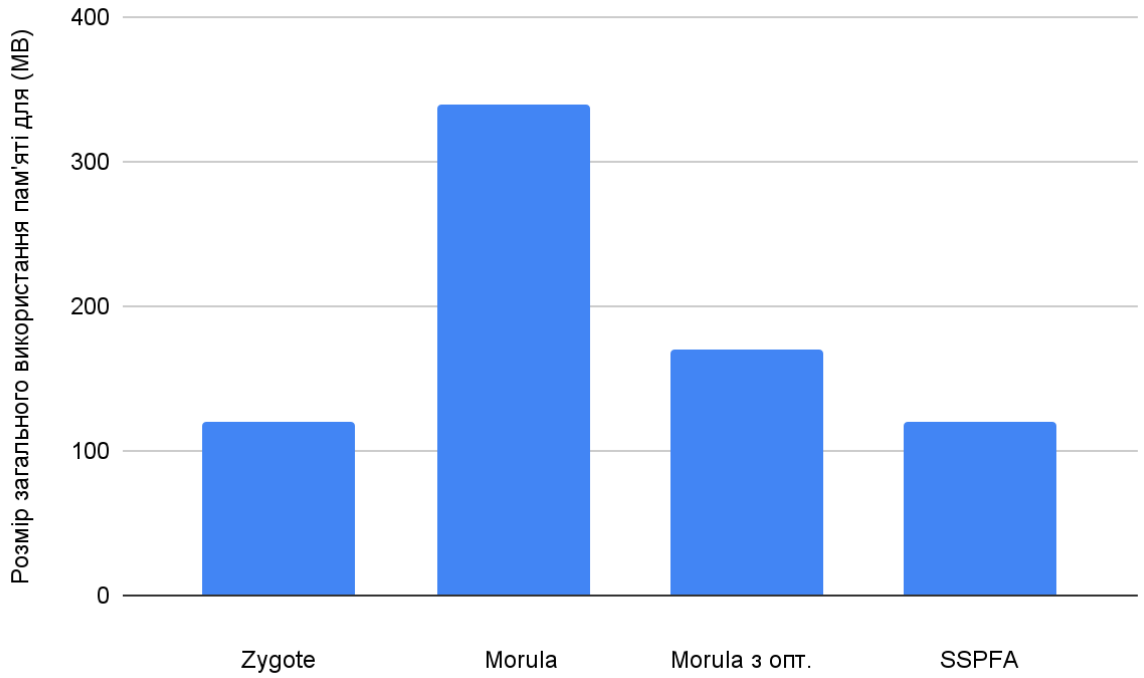

Рисунок 4.5 — Використання пам’яті для завантаження пристрою

З діаграми бачимо що для накладних витрат на використання пам’яті, модифікації використали: Morula — 283% (210 МБ), Morula з оптимізацією — 141%(50 МБ), SSPFA — до 1%(25 байт). Така велика різниця у першому випадку пояснюється тим, що Morula не дозволяє ділитися такими ресурсами, як переміщені розділи або купи, виділені DVM. Майже нульові накладні витрати на пам’ять бачимо, при застосування SSPFA, це відбувається завдяки тому що функція знаходиться в спільній бібліотеці libdvm.so, через те розмір програми майже не збільшується в процесорі ARM.

Для оцінки впливу на продуктивність виконання програм, було обрано 5 додатків з Google Play: Twitter, Skype, Instagram, Telegram, TikTok. Кожна програма виконується за наступними кроками: пристрій Android завантажується з моделлю створення процесу Zygote, далі встановлюється властивість системи, яка визначає, яка модель створення процесу (Zygote/Morula) буде використовуватися для виконання програми, для SSPFA звичайний алгоритм створення процесу після завантаження. Потім програма автоматично виконується шляхом надсилання

Intent до основного класу активності через інтерфейси терміналу adb, що має той самий ефект, що й користувач, натискаючи значок програми, щоб запустити програму. В даному випадку модель створення процесу вказується після завантаження пристрою, щоб уникнути впливу будь-яких загальносистемних ефектів під час процесів завантаження.

Для додатків вимірюється час запуску — час, що минув між часом, коли менеджер діяльності отримує запит на створення програми, і часом, коли цільова програма відображається на екрані, дані отримані з повідомлень журналу записаних через інтерфейси терміналів adb та використання пам'яті — private dirty для програми з /proc/pid/smaps, який показує обсяг RAM, який не використовується жодним іншим додатком.

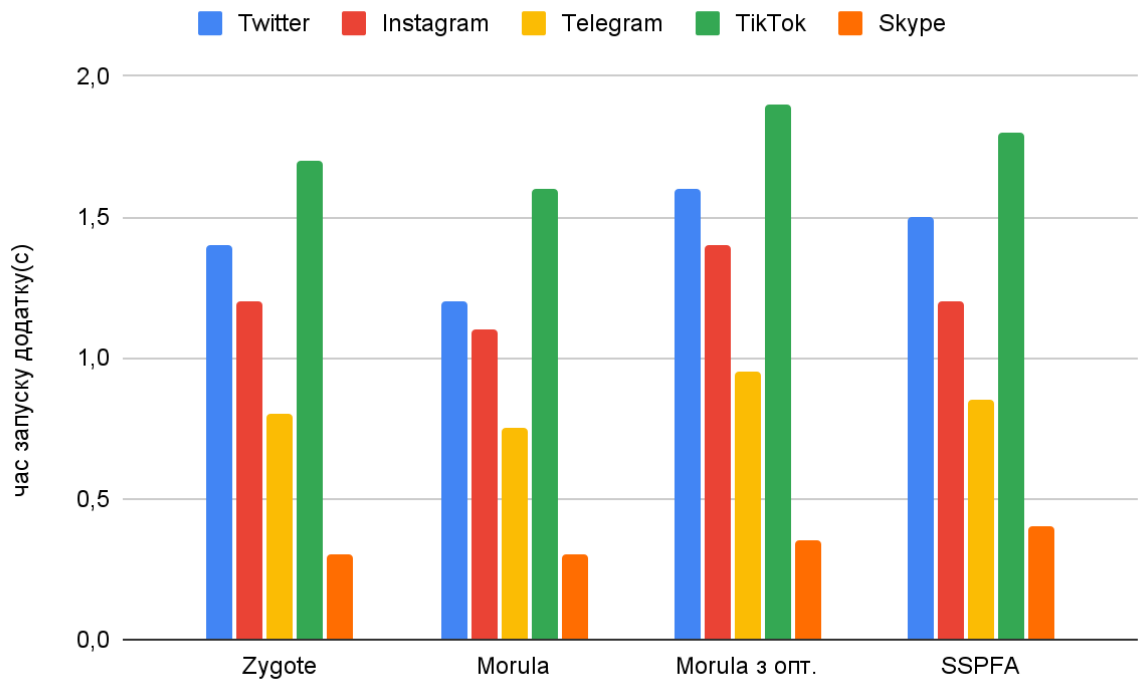

Рисунок 4.6 — Час запуску додатків при різних моделях

З Рисунку 4.4 бачимо що, для всіх п'яти оцінених додатків Morula і Zygote показують майже однаковий час запуску. У чотирьох додатках дана модель була швидшою на ~1%. Це також означає, що інструмент Morula завчасно обчислив

ініціалізацію ресурсу та переміщення адреси, і ці обчислення не були включені в час запуску. Коли на Morula застосовується завантаження на вимогу(оптимізація), час запуску збільшується на 11% - 20% порівняно з Zygote, це уповільнення пов'язане з тим, що класи Dalvik необхідно завантажувати під час запуску. SSPFA є повільнішою за Zygote у чотирьох випадках на ~6%, при запуску Instagram, їх час рівний.

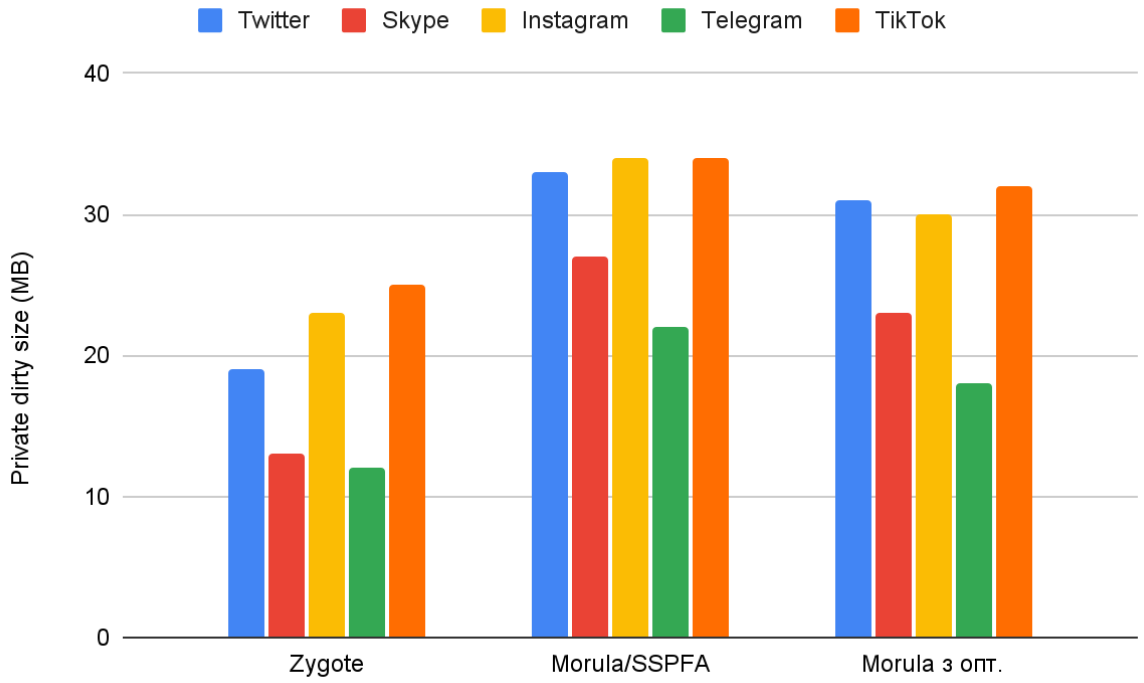

Рисунок 4.7 — Розмір Private dirty при різних моделях

На Рисунку 4.5 порівняно пам'ять при трьох моделях, так як при SSPFA вона рівна Morula. У середньому Morula використовує на 11,6 МБ більше пам'яті в порівнянні з моделлю Zygote. Ці витрати пов'язані з переміщенням адрес, через рандомізацію макетів адрес, усі покажчики відносної адресації змінюють своє розташування та приватною купою DVM. Коли ми застосовуємо оптимізацію завантаження на вимогу, Morula використовує в середньому на 8,4 МБ більше пам'яті, що на 3,2 МБ менше, ніж Morula до оптимізації, ця різниця є результатом незавантаження непотрібних класів Dalvik.

Висновки до розділу 4

В даному розділі надано технічний опис тестового середовища, оцінено ефективність методів та їх вплив на продуктивність ОС Android.

З аналізу бачимо, що найменший вплив на час завантаження телефону має інструмент Morula з оптимізацією, водночас метод SSPFA майже не впливає на використання пам’яті. Morula має перевагу над Zygote, проте використовує найбільше пам’яті при роботі з додатками.

Отже, найоптимальнішим варіантом є інтеграція SSPFA, так як вона майже не має навантаження на пам’ять, а додаткові витрати часу не є критичними. Також, можемо розглянути Morula без оптимізації, даний інструмент виграє у SSPFA різницю у часі з Zygote у роботі з додатками, так як швидкість для користувача у сучасному світі є надважливим критерієм.

ВИСНОВКИ

Головною метою даної дипломної роботи було запропонувати та протестувати метод для запобігання Code Execution вразливостей в ОС Android.

Протягом роботи було проаналізовано будову ОС Android, алгоритм створення процесів та природу виникнення вразливостей. В результаті даного дослідження було виявлено, що більшість атак RCE можна проводити завдяки вразливостям у роботі з пам’яттю, що дозволяє зловмиснику користуючись ними отримувати доступ до інформації.

Для пошуку рішення захисту, було опрацьовано поточні методи запобігання даного типу вразливостей, які покривають різні види аналізу коду та вбудовані механізми в ОС Android такі як ASLR. В ході дослідження виявлено, що останній метод має певний конфлікт із процесом створення процесу завантаження за допомогою моделі Zygote — не відбувається повна рандомізація пам’яті. Та метод захисту від переповнення буферу SSP не є дієвим, якщо зловмисник вже отримав канаркове значення.

Розглянуто методи для пом’якшення вищеописаних проблем —- Morula та SSPFA. Описано алгоритм їх дії та зазначено, що вони не змінюють логіку поточного створення процесу.

Реалізовано та протестовано запропоновані методи запобігання Code Execution атакам, які відбуваються через неправильну перевірку меж масиву або витік канаркового значення. Проведено оцінку їх впливу на продуктивність ОС Android та ефективність виконання поставлених задач, що оцінювалося за допомогою вимірювання часу запуску програм та використовуваної пам’яті.

Дана тема потребує подальших досліджень та тестувань методів на різних пристроях ОС Android для повної оцінки їх реалізації в перспективі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Статистика користувачів ОС[Електронний ресурс] – 2022. – Режим доступу: https://gs.statcounter.com/os-market-share/mobile/worldwide
2. Denis Crăciunescu. Про рівні безпеки ОС Android [Електронний ресурс] – 2020. – Режим доступу: https://proandroiddev.com/the-layers-of-the-android-security-model-90f471015ae6
3. Статистика вразливостей ОС Android[Електронний ресурс] – 2022. – Режим доступу: https://www.cvedetails.com/product/19997/?q=android
4. Tsviatko Yovtchev. Загальний опис RCE атак[Електронний ресурс] – 2021. – Режим доступу: https://blog.sqreen.com/remote-code-execution-rce-explained/
5. Опис RCE вразливостей ОС Android[Електронний ресурс] – 2022. – Режим доступу: https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/year-2022/opec-1/Google-Android.html
6. Amrita Pathak. RASP інструменти [Електронний ресурс] – 2022. – Режим доступу: https://geekflare.com/rasp-tools/
7. Prabhaker Mateti: Towards an Enhanced Secure Android[Електронний ресурс] – 2017. – Режим доступу: https://www.researchgate.net/publication/320952677_ASLR_and_ROP_Attack_Mitigations_for_ARM-Based_Android_Devices
8. Опис інструменту Scudo [Електронний ресурс] – 2022. – Режим доступу: https://source.android.com/devices/tech/debug/scudo
9. Linda Rosencrance. Опис принципу останніх привілей – 2021. – Режим доступу: https://www.techtarget.com/searchsecurity/definition/principle-of-least-privilege-POLP

10. Опис інструменту ShadowCallStack[Електронний ресурс] – 2022. – Режим доступу: https://source.android.com/devices/tech/debug/shadow-call-stack
11. Опис інструменту HWAddressSanitizer[Електронний ресурс] – 2022. – Режим доступу: https://source.android.google.cn/devices/tech/debug/hwasan
12. Опис інструменту WARNING_PATH_SENSITIVE_USE_GSL_AT [Електронний ресурс] – 2021. – Режим доступу:https://devblogs.microsoft.com/cppblog/new-static-analysis-rule-for-bounds-checking/
13. Опис інструменту BoundsSanitize[Електронний ресурс] – 2021. – Режим доступу: https://www.google.com/url?q=https://source.android.com/devices/tech/debug/bounds-sanitizer&sa=D&source=docs&ust=1654711415277364&usg=AOvVaw0j6WmXaHZG5vC3BlXChvtG
14. Опис інструменту Morula[Електронний ресурс] – 2014. – Режим доступу: https://www.google.com/url?q=https://github.com/lifeasageek/morula&sa=D&source=docs&ust=1654711415270862&usg=AOvVaw0cvPzyr8rpSV3WMwvEI3fI
15. Byoungyoung Lee, Long Lu , Tielei Wang, Taesoo Kim , Wenke Lee:From Zygote to Morula[Електронний ресурс] – 2014. – Режим доступу: https://wenke.gtisc.gatech.edu/papers/morula.pdf
16. Héctor Marco-Gisbert, Ismael Ripoll-Ripoll: SSPFA: effective stack smashing protection for Android OS[Електронний ресурс] – 2019. – Режим доступу: https://d-nb.info/1179057201/34