Aristotle University of Thessaloniki
Faculty of Sciences
School of Informatics
Master of Communication Networks and Systems Security

Vulnerability discovery in Android System through fuzzing

by
Athanasios Oikonomou

Supervisor: Petros Nikopolitidis
Associate Professor AUTH

April 2, 2023

Abstract

Finding vulnerabilities in Android system and applications is a challenging task. The success of fuzzing on Windows/Linux/Mac OS programs makes one wonder how effective it would be to apply these tools to Android. After all, Android is just a modified Linux kernel with additional utilities to support special hardware components like touchscreens and GSM antennas. The effectiveness of fuzzing depends on the level of identifying Android's attack surface and proper data mutation. The majority of Android applications are closed source so finding a way to access inner available inputs can be very tricky. Also, fuzzing as a procedure is computationally very intensive so new ways and techniques must be found. In this thesis, we will look at how fuzzing tools are working, various aspects that can affect the performance of such a tool, how the Android system is built and what of its components are crucial for attacking it. Next, we will present our developed tool that overcomes many of the known limitations of fuzzing Android applications. This tool is based in Frida reverse engineering framework, giving as the advantage of dynamic instrumentation overcoming in that way possible authentication mechanisms that may be implemented into the application, the advantage of easily modifying parts of fuzzer adapting in this way at possible application's singularities. Having in mind the limitation of device's hardware performance the tool designed in such way that can be run in virtual Android devices with increased memory and CPU capabilities and capable of running in parallel with different data mutation approaches in each device reducing the time of finding vulnerabilities.

Vulnerability discovery in Android System through fuzzing

Athanasios Oikonomoy
athaoiko@csd.auth.gr

April 2, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Android popularity

The use of the Internet has increased dramatically in recent years, as has our reliance on it. Hundreds of services that were once normal are now digital and play an important role in our lives. People use mobile devices to access online services more than ever before, so most people carry a fully networked computing device in their pocket. Also, with the growth of the Internet of Things (IoT), we must add to these devices smart things like our refrigerator and our home's security system. According to statistics by Statcounter Global Stats [6] also shown in Figure 1.1, the majority of these mobile devices (and IoT devices that not shown in figure) running the Android Systems. Although it is designed for smartphones, now can be found in laptops, tablets, TVs, wearable devices, GPS trackers, cars and many more types of devices. Since its first release to market in 2008, Google's Android operating system has been a huge success, drastically eclipsing the market share of any other mobile operating system. As a result, the security of these linked devices is becoming increasingly critical. So, when we talk about security, we talk about for flaws and vulnerabilities in the foundation mechanism of Android System: system's infrastructure, included or third-party installed applications, networks protocols libraries and other. Detecting flaws on them must be quick and reliable, which is impossible to accomplish without the use of specialized automation technologies.

According to Google, Android devices will receive security updates for three years after their release, and a couple of years after that by the vendors. If we take into account the statistics by Statcounter Global Stats, as shown in Figure 1.2 there are already millions of devices in the global market that have not received security updates or patches for years. And, as shown in Figure 1.3 these devices still exist in period of two years without significant change.

Google's Vise President of Product Management (Android & Google Play) Sameer Samat at his speech in Google I/O 2022 conference confirmed that over 1 billion new Android devices were activated in just the last year. These about 1.5 million of new devices are being activated every day. This is a huge number considering there are only 8 billion people on the planet. This means that an eighth of the world's population activated an Android phone from any manufacturer within the past 365 days.

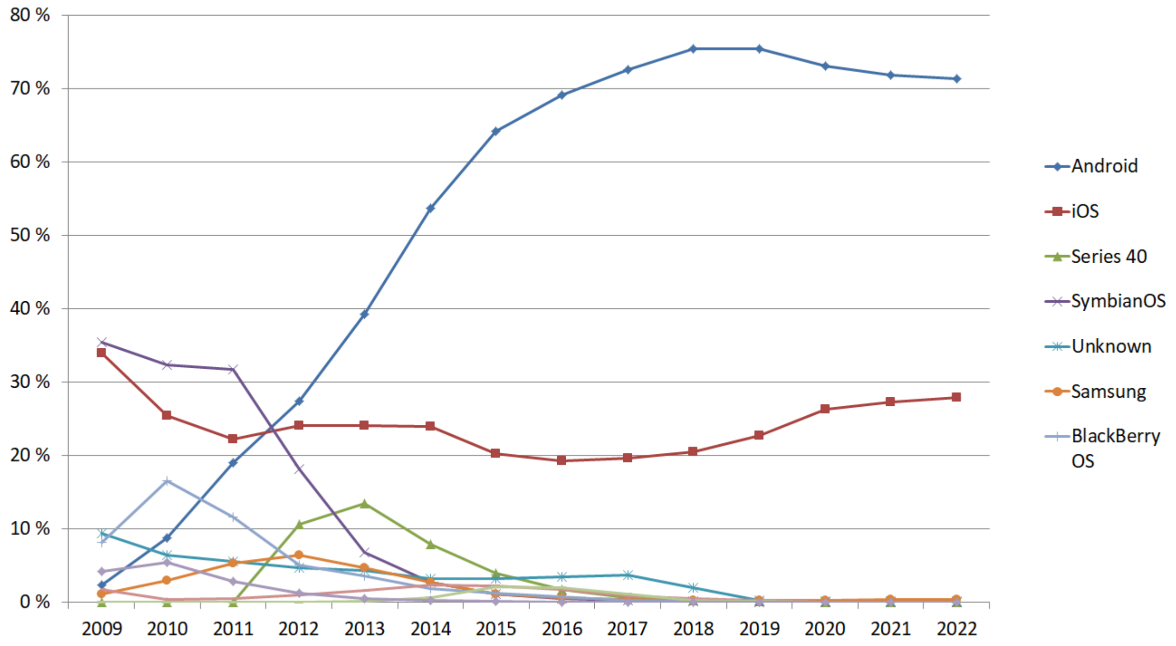


Figure 1.1: Mobile OS popularity.

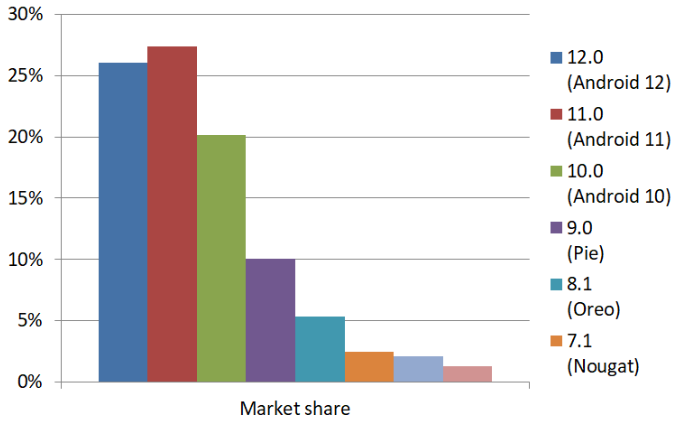


Figure 1.2: Android versions market share Sep 2022.

The vast number of unique devices and users that utilize the Android operating system makes it an ideal target for cyber criminals. A single discovered flaw has the potential to affect thousands of users and their data. Another reason that the Android system is such an attractive target is that new versions with new features are developed every few months. This necessitates a lot of new code, which often contains vulnerabilities. It should be noted that finding and addressing these vulnerabilities requires a significant amount of time and resources, which are not always available.

1.2 Vulnerabilities' increase

According to Zimperium's Global Mobile Threat Report 2022 [7] over 30% of known, zero-day vulnerabilities discovered in 2021 targeting mobile devices. In figures 1.4 and 1.5, the presented statistics from Google [8] about Common Vulnerabilities and

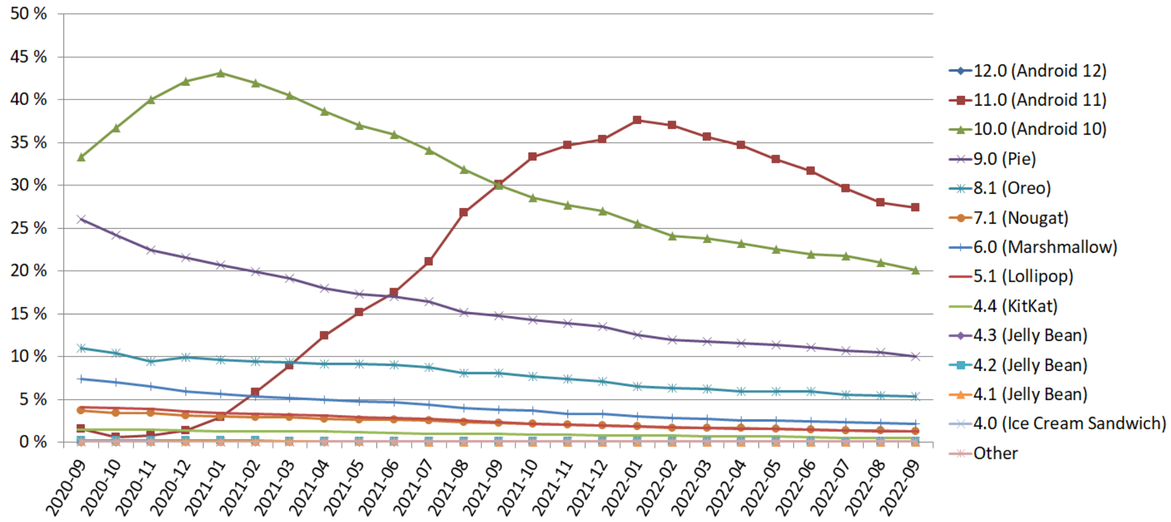


Figure 1.3: Android versions market share between Sep 2020 and Sep 2022.

Exposures (CVE) that found in android system revealing a significant increase in zero-day exploits in smartphones and tablets. By observing the most common types of vulnerabilities, we can assume that the majority of them (buffer overflows and code executions) are caused by native code that is used by the Android system and applications as third-party libraries. We will discuss this more in next chapters. It is remarkable that 21% of CVEs are categorized with a medium attack complexity, while 79% are categorized with a low attack complexity. 23% of the tracked CVEs rated a Common Vulnerability Scoring System (CVSS) [9] score of 7.2 or higher, with 2.1% are falling into the critical category.

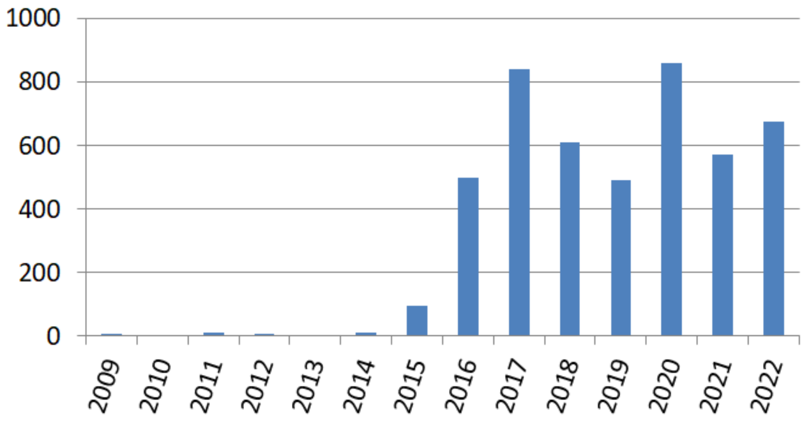


Figure 1.4: Number of android's vulnerabilities per year.

Also there is a interesting list [10] and maintained by Google that contains information for zero-day exploits in the wild. By the term in the wild Google means the vulnerabilities that detected in actual attacks against users. We must not that, for all those vulnerabilities that the time the attack took place there was not any available patch for the devices. In this list we observe two thinks. Firstly, the total number of zero-day vulnerabilities doubled in one year something that indicates the increased interest of malware activity and secondly, there is a 466% increase in zero-day vulnerabilities that used in attacks against mobile devices in 2021 contrary to previous

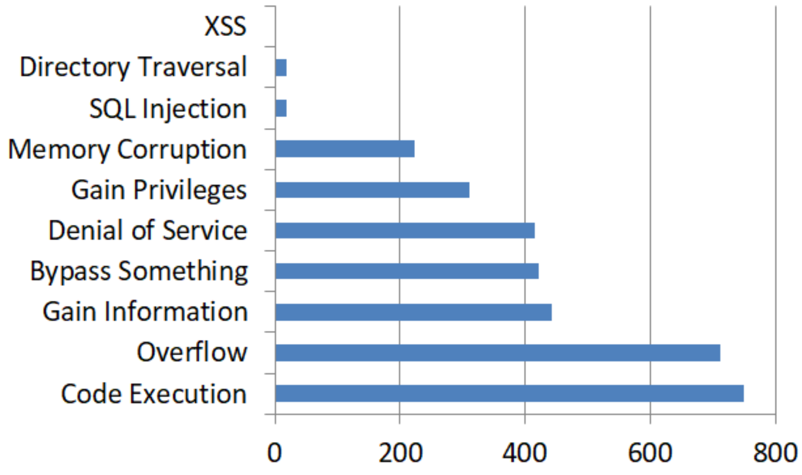


Figure 1.5: Number of android's vulnerabilities by type.

years. According to recent estimates, 95% of all mobile malware today is designed to target Android devices, and 73% of that malware is created specifically to generate profits.

Useful statistics from this list are the following:

- A new zero-day exploit is discovered every 17 days.
- It needs in average 15 days from the vendors to develop and public a new patch for the specific zero-day vulnerability.
- It is published only for the 86% of found vulnerabilities a technical analysis and details for the causes of them.
- The most usual cause of them are the memory corruption at 68% of listed CVEs.

This list does not include all zero-day vulnerabilities but only that published and in different sources and was interesting for research purposes by Google's Project Zero team.

The National Vulnerability Database (NVD) maintained by the National Institute of Technology (NIST) has interesting statistics that are shown in Figure 1.7. It is not a coincidence that after 2014, when android popularity increased significantly (Figure 1.1), detected vulnerabilities also increased. Although mobile vendors, Google and developers are becoming more and more concerned about privacy and protecting their users' personal information are working hard to address these issues vulnerabilities still exist and are many. As long as there are faults in systems and applications, the possibilities to exploit them and compromise users' personal data are also increased.

Remarkable is the fact that there are companies that have bounties programs especially for mobile zero-day exploits and pay for each one from $2,500 to $2,500,000. In Figure 1.8 we can see the amounts paid by Zerodium [1] to researchers to acquire original zero-day exploits depending on the popularity and security level of the affected software/system and the quality of the submitted exploit. The quality of the

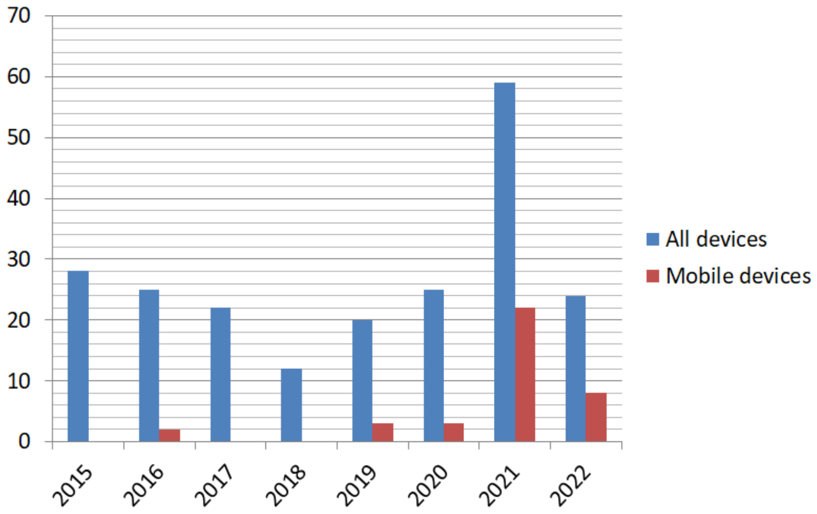


Figure 1.6: Number of zero-day exploited vulnerabilities found in the wild.

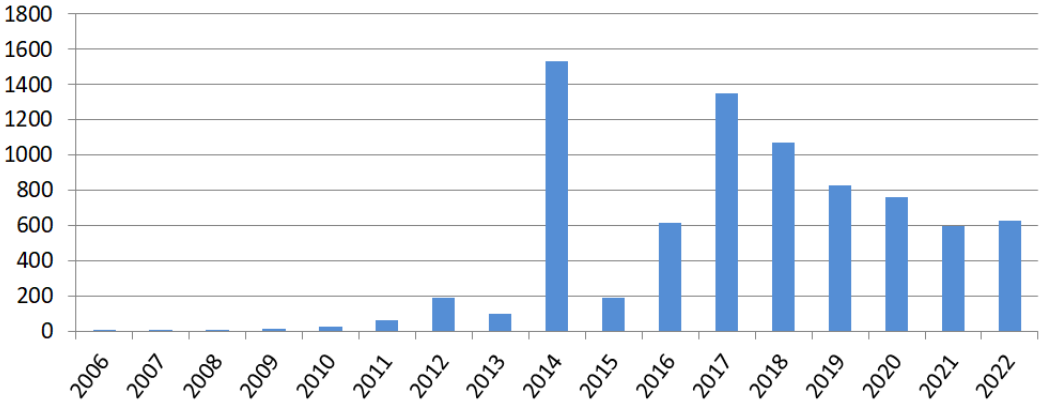


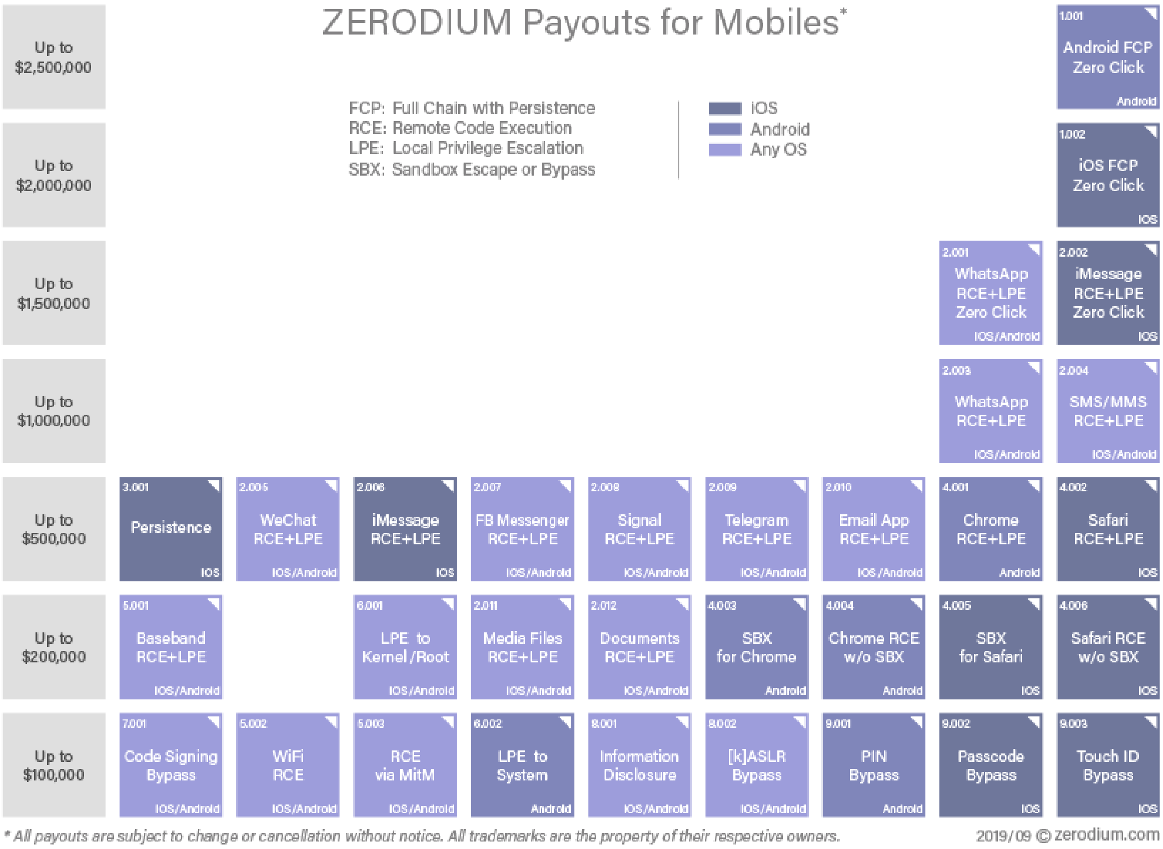
Figure 1.7: Number of android vulnerabilities from NIST database.

exploit has to do with the number of supported versions/systems/architectures, reliability, bypassed exploit mitigations, default vs. non-default components, process continuation and other.

1.3 Google's philosophy

Google having in mind the goal of increased security and privacy of Android system according to the documentation [11] follows a robust security model making things easy for developers and end-users:

- Android was **designed to be open** allowing access to different types of hardware and software and data available remotely or locally. In order this open characteristic to be supported is provided by Google a environment responsible for the confidentiality, integrity, and availability of users, data, applications, the devices, and the network.
- Android was **designed for developers** by providing to them all the needed security mechanisms reducing the burden of developing. It offers a stable platform, a security team responsible for searching vulnerability and providing



ZERODIUM Payouts for Mobiles*

FCP: Full Chain with Persistence
RCE: Remote Code Execution
LPE: Local Privilege Escalation
SBX: Sandbox Escape or Bypass

▬ iOS
▬ Android
▬ Any OS

Up to $2,500,000										1.001 Android FCP Zero Click Android
Up to $2,000,000										1.002 iOS FCP Zero Click iOS
Up to $1,500,000									2.001 WhatsApp RCE+LPE Zero Click iOS/Android	2.002 iMessage RCE+LPE Zero Click iOS
Up to $1,000,000									2.003 WhatsApp RCE+LPE iOS/Android	2.004 SMS/MMS RCE+LPE iOS/Android
Up to $500,000	3.001 Persistence iOS	2.005 WeChat RCE+LPE iOS/Android	2.006 iMessage RCE+LPE iOS	2.007 FB Messenger RCE+LPE iOS/Android	2.008 Signal RCE+LPE iOS/Android	2.009 Telegram RCE+LPE iOS/Android	2.010 Email App RCE+LPE iOS/Android	4.001 Chrome RCE+LPE Android	4.002 Safari RCE+LPE iOS
Up to $200,000	5.001 Baseband RCE+LPE iOS/Android		6.001 LPE to Kernel/Root iOS/Android	2.011 Media Files RCE+LPE iOS/Android	2.012 Documents RCE+LPE iOS/Android	4.003 SBX for Chrome Android	4.004 Chrome RCE w/o SBX Android	4.005 SBX for Safari iOS	4.006 Safari RCE w/o SBX iOS
Up to $100,000	7.001 Code Signing Bypass iOS/Android	5.002 WiFi RCE iOS/Android	5.003 RCE via MitM iOS/Android	6.002 LPE to System Android	8.001 Information Disclosure iOS/Android	8.002 [k]ASLR Bypass iOS/Android	9.001 PIN Bypass Android	9.002 Passcode Bypass iOS	9.003 Touch ID Bypass iOS

* All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.

2019/09 © zerodium.com

Figure 1.8: Zerodium prices for zero-day exploits. [1]

advises for secure coding, Google Play Market and Play services mechanisms to provide security libraries, updates and fixes to all devices and users.

- Android was **designed for users** giving them the possibility to see and control all application's needed permissions prohibiting from attacks like social engineering attacks.

At timing writing this thesis Android is at version 12. Every new release since 2008 the day than Android made public has improved security and user control, while trying to maintain a balance between the two. This balance is responsible for Android security mechanisms that can be removed after some releases. Now days Android has become less vulnerable to common exploit techniques, such as buffer overflows. In addition, its application isolation mechanism, sandboxing, has been strengthened, and the number of system processes that run as root has been reduced significantly, further reducing the potential attack surface.

Following some of the must important security enhancements per Android version, from 1.5 through 11:

Android 1.5

Stack buffer oveflow protector (ProPolice), safe_iop to reduce integer overflows, OpenBSD dlmalloc to prevent from double free(), OpenBSD calloc to prevent integer overflows during memory allocation.

Android 2.3

Format string vulnerability protections, Hardware-based No eXecute (NX) to prevent code execution on the stack and heap, Linux mmap_min_addr to mitigate null pointer dereference privilege escalation.

Android 4.0

Address Space Layout Randomization (ASLR) to randomize key locations in memory

Android 4.1

PIE (Position Independent Executable) support, Read-only relocations / immediate binding, dmesg_restrict and kptr_restrict enabled to avoid leaking kernel addresses.

Android 4.2

Application verification, prior to installation, more control of premium SMS, Always-on VPN, Certificate Pinning protecting from possible compromise of Certificate Authorities, Improved display of Android permissions, installd hardening (not running as root user), init script hardening, ContentProvider default configuration, enhanced Cryptography libraries. Security Fixes for open source libraries with security fixes include WebKit, libpng, OpenSSL, and LibXML.

Android 4.3

Android sandbox reinforced with SELinux, No setuid/setgid programs, ADB Authentication, Restrict Setuid from Android Apps, AndroidKeyStore Provider allowing applications to create exclusive use keys, KeyChain isBoundKeyAlgorithm allowing applications to confirm system-wide keys, NO_NEW_PRIVS preventing applications from performing operations which can elevate privileges via execve function, Relocation protections, FORTIFY_SOURCE enhancements supporting strchr(), strrchr(), strlen(), and umask() calls, Improved EntropyMixer writing entropy at shutdown/reboot and Security Fixes for Android-specific vulnerabilities.

Android 4.4

Android sandbox reinforced with SELinux (enforcing mode enabled), Per User VPN, ECDSA Provider support in AndroidKeyStore, Device Monitoring Warnings, FORTIFY_SOURCE supporting level 2 and clang compiler, Certificate Pinning preventing from fraudulent Google certificates and Security Fixes for Android-specific vulnerabilities.

Android 5.0

Full disk encryption enabled by default with user password protection using scrypt, Android sandbox reinforced with SELinux for all domains, Smart Lock using trustlets, Multi user, restricted profile, and guest modes for phones and tablets, Updates to WebView without OTA, Updated cryptography for HTTPS and TLS/SSL (enabled TLSv1.2 and TLSv1.1), non-PIE linker support removed enhancing this way Android's ASLR, FORTIFY_SOURCE improvements (supports stpcpy(), stpncpy(), read(), recvfrom(), FD_CLR(), FD_SET(), and FD_ISSET() calls) and security Fixes for Android-specific vulnerabilities.

Android 6.0

Runtime Permissions, Verified Boot, Hardware-Isolated Security, Fingerprints, SD Card Adoption, Clear Text Traffic, System Hardening and USB Access Control.

Android 7.0

File-based encryption, Direct Boot, Verified Boot (now strictly enforced), SELinux (Updated configuration), Library load-order randomization and improved ASLR, Kernel hardening, APK signature scheme v2, Trusted CA store and Network Security Config.

Android 8.0

Encryption, Android Verified Boot (AVB), Lock screen credentials, KeyStore, Sandboxing using Project Treble's standard interface, Kernel hardening, Userspace hardening, Streaming OS update, Install unknown apps and Privacy using different value of Android ID (SSAID) for each app and each user.

Android 9

Generic system image (GSI), HIDL framework backwards compatibility, Dynamically available Hardware abstraction layers (HALs), Compressed tree overlays, device tree blob overlay (DTBO) verification, Vendor Native Development Kit (VNDK) tools, application binary interface (ABI) checker, VNDK snapshots, Canonical boot reason compliance, System as Root, DTBO in recovery, Synchronized app transitions, Text classification service, Wide-gamut color support, Compatibility Test Suite (CTS) downloads, Sensor fusion box for camera, Service name-aware HAL testing, Advanced telemetry for debugging, Biometric support, Control flow integrity (CFI) for combiled binaries and kernel, File-based encryption (FBE) with adoptable storage, Metadata encryption, StrongBox, 3DES support, Android Protected Confirmation API, Per-app SELinux sandbox, External USB cameras, Multi-camera support, Motion tracking, Carrier identification, eSIM support, Multi-SIM support for IP Multimedia Subsystem (IMS), MAC randomization, Wi-Fi round trip time (RTT), WiFiStateMachine improvements, WPS deprecation, WinScope tool for window transition tracing, Vehicle HAL, global navigation satellite system (GNSS) HAL, Whitelisting privileged apps permissions, Bandwidth estimation improvements, eBPF traffic monitoring tools, Managed profile improvements, APK caching, Write-ahead logging (WAL), Background restrictions and Batteryless devices.

Android 10

BoundsSanitizer in Bluetooth and codecs, Execute-only memory, Extended access, Face authentication, Integer Overflow Sanitization, OEMCrypto API, Scudo, ShadowCallStack, WPA3 and Wi-Fi Enhanced Open support, Background activity restrictions, Camera metadata, Clipboard data protection, Device location permissions, scoped external storage protection, MAC address randomization by default, /proc/net filesystem restrictions, Restricted access to screen contents and Restrictions on direct access to configured Wi-Fi networks.

Android 11

Boot header version 3, Boot partitions, Vendor boot header more than one page, Recovery images, Android common kernels, Android kernel ABI Monitoring, Kernel Module support, DebugFS remove, ION heaps for Generic Kernel Image (GKI), Auto Revoke Permissions, Runtime resource overlays, Audio capture from FM tuner requires a privileged permission, USB Port Reset API, Camera bokeh support, Improved camera support for Android virtual devices, NFC off-host payment synchronization, Quick Access Wallet, eSIM activation flow through carrier app, eUICC API error handling, Multi-operator network support, Small cell support, Open Mobile API changes, Wi-Fi hotspot support for tethering, Wi-Fi Passpoint enhancements, Data access auditing, Implementing Custom User Types, Context Hub Runtime Environment updates, Gamepads support, Neural Networks support, Hinge angle sensor type, Userspace lmkd, Scoped storage support, SDCardFS Deprecation, CTS tests for APEX management APIs, Scoped vendor logging, GWP-ASan: heap corruption detection, OTA packages for multiple SKUs, Virtual A/B, Scudo heap allocator by default and TV Input Framework support.

1.4 About this thesis

Firstly, in Section 2 we will see how fuzzing works, what the best practices are and what is needed depending on our type of target program. The big picture of this tools help as to identify which parts of protocols structure are important for our cause. Later in Section 3, we will look how the Android system and it's applications are working. During this section only the properties are relative and will play significant role in fuzzing will be mentioned. For a successful vulnerability discovery data generation is critical, so we must understand not only how computers use protocols but also their structure. So, identifying Android's structure and different potential inputs we can adjust our fuzzing approach. In Section 4, we will analyse our approach in developing our fuzzer and demonstrate a scenario of fuzzing a Android application that we developed with memory related vulnerabilities intentionally. Finally, the paper is concluded in Section 5.

Chapter 2

Android

Android is an operating system (OS) designed to run on mobile phones, tablets and smart devices like smartwatches, TVs and home assistants. It is based on a modified Linux kernel and hundreds of other open source utilities targeting touchscreen capable devices. Table 2.1 shows the core android projects. It is developed mainly in Java at 40%, C at 18% and C++ at 19%. In general Android can be divided in five layers. From top to bottom, there are the applications built in Java programming language using the Android Software Development Kit (SDK), the Android framework, the Dalvik Virtual Machine (DalvikVM) which is included in Android Runtime (ART), native-code and Linux kernel. The combination of its open-source nature, amazingly big number of available applications to install and large development community made Android the most famous mobile operating system. Android Package Kit (APK) is a file format used in distributing and installing Android applications (APP). It contains all the necessary files that are required for the installation.

Project	**Description**
bionic	C runtime: libc, libm, libdl, dynamic linker
bootloader/legacy	Bootloader reference code
build	Build system
dalvik	Dalvik virtual machine
development	High-level development and debugging tools
frameworks/base	Core Android app framework libraries
frameworks/policies/base	Framework configuration policies
hardware/libhardware	Hardware abstraction library
hardware/ril	Radio interface layer
kernel	Linux kernel
prebuilt	Binaries to support Linux and Mac OS builds
recovery	System recovery environment
system/bluetooth	Bluetooth tools
system/core	Minimal bootable environment
system/extras	Low-level debugging/inspection tools
system/wlan/ti	TI 1251 WLAN driver and tools

Table 2.1: Core Android Projects.

2.1 System architecture

For developers, Android's OS is the most important layer between applications and the hardware. Even the simplest applications in order to function properly need access to hardware like touch screen, WiFi antenna etc.

OS is responsible for three main things:

- Hardware management on behalf of applications.
- Service management by providing or not access to resources like networking and memory.
- Applications execution.

Figure 2.1 shows the logical architecture of the Android OS. At the lowest level we can see the modified **Linux kernel**. Earlier versions of Android, before 4.0 used the Linux kernel 2.6.x, while the newer versions are using version 3.x. As in any Unix system, the main purpose of kernel is to provide special drivers for interaction with the hardware. But in Android is a bit different than that we can find in our desktop computer running a normal Linux distribution like Ubuntu or Fedora. The difference has to do with special functions needed by Android system like special security mechanism (application's permissions) or special hardware like touchscreens. Management of CPU resources, management of system memory and application execution are parts of its processes. Thanks to his design, it can be run on all kind of devices from watches to heavy load servers.

On top of the Linux kernel there is the **Hardware Abstraction Layer (HAL)**. HAL is a collection of interfaces that expose capabilities of hardware devices to higher layers. The HAL using native library modules implements interfaces for every type of available hardware component like camera or Bluetooth. When an application makes a call to access a hardware device through framework API, the Android OS firstly loads the appropriate library module for that hardware component and then gets access to that component.

The core libraries of Android system are developed in Java, but of course they are a lot of native coded. So, above HAL exist these **Native libraries** like SQLite and OpenGL that are written mostly in C/C++ and **ART** which contains the Java core libraries. These layer includes all these libraries that are being used for application development. ART is the application environment runtime which is needed by applications to run. The Java runtime libraries can accessed and called directly by the system and applications but native libraries needs the Java Native Interface (JNI). JNI allows java code to call native and vise versa. Also, thanks to ART each application is running in its own process and with its own instance of the ART. In more depth, ART runs multiple virtual machines one for each application and executes DEX files. DEX uses a special bytecode format designed specifically for Android OS. All those core Java libraries until Android 7.0 (Nougat) are derived from the Apache Harmony Project [12]. From Android 8.0 (Oreo) derived from OpenJDK.

Above them is the **Java API Framework Layer**. By looking the architecture diagram web can observe that as a layer is above of both of the low level native libraries and ART. This is the layer that developers will interact with as it contains all the neces-

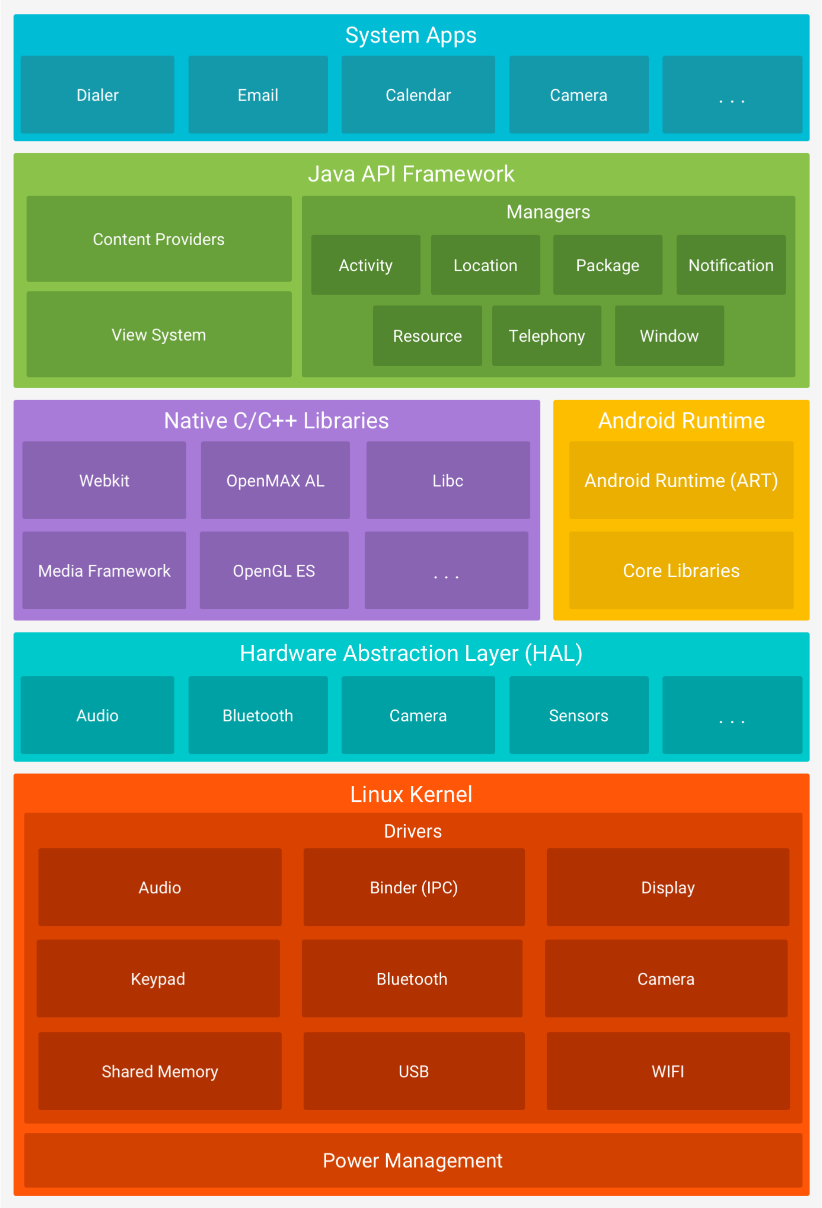


Figure 2.1: Android Architecture. [2]

sary libraries and system services that are needed for writing applications. System services are implement most of the core Android systems features like touchscreen support or network connectivity. These services are a kind of portable and can be called from other services or applications. The inter-connection between services or other Android components as all of them are running isolated from the environment (separate address spaces) is possible thanks to **Inter-Process Communication (IPC)** mechanism provided by **Binder** which is responsible for service discovery and message exchange. IPC is similar to Unix's shared files, signals, sockets, pipes, semaphores, shared memory or message queues solutions. Binder is a new implementation special designed for use in Android. It is based on the architecture and ideas of OpenBinder [13]. Binder is similar to Windows' Common Object Model (COM) and Unix's Common Object Broker Request Architectures (CORBA) without supporting remote procedure calls (RPC) across the network.

Finally, on top of all layers is the **Application Layer**. This layer is the place where all applications exists including prebuilt apps like Dialer, Camera by manufacturer and all those that are developed by third parties and installed later by the user through Play Store.

2.2 Applications

Android applications are the programs that user can interact with and developed using the **Java Development Kit (JDK)** and compiled into bytecode or using the **C/C++ Native Development Kit (NDK)** and compiled into native binary code (running directly on CPU). There are two types, system applications and third-party user installed applications.

System applications are preinstalled and part of the Android core system included in the OS's primary image without giving the option to users to remove theme or changed. These applications are considered secure and have more privileges than user installed applications. Examples of such application are camera, messages, Google Play Store, drivers or browsers shipped by the vendors. There are three types: Google Apps, Core Modules Apps and Bloatware. These **Google Apps** are Google Search Engine, Google Chrome, Gmail, Google Maps and more depending on the signed the Anti-Fragmentation Agreement (AFA) between smartphone manufacturers companies and Google. **Core Module Apps** are the critical important applications that are needed from Android system to function properly. Some of them are, com.android.bluetooth, com.android.phone, com.android.keychain and com.android.nfc. **Bloatware** are applications that are installed by Smartphone manufacturers in order to modify the Android system according to their needs. Examples, are custom UI launchers or backup services. Preinstalled and bloatware system applications are stored under read-only /system/app folder and system applications that require privileged access and signed with the platform signing key are under read-only /system/priv-app folder.

User installed applications also called Non-System Apps or third-party apps are programs that developed by companies or independent developers and can be installed from Google Play Store. These are installed under a read-write /data/app folder and can be uninstalled at will by the users. Each application is running in a sandboxed and isolated environment having access only to resources and data that they have permissions to. Thanks to this isolated environment access to other application's data is forbidden.

As mentioned earlier Android system and its application are implemented in Java and this is why a Java Virtual Machine (JVM) is needed to execute them. Android's specific JVM implementation is called **Dalvik VM**. After 2016 with the Android version 6 Dalvik is included in ART and doesn't mentioned separately. Dalvik VM or later ART can not execute java's .class files but uses a custom file format, the Dalvik Executable (.dex file). So in case of JDK, it is used the special bytecode optimized .dex format. Dalvik VM was developed by Google with efficiency and security in mind. Dalvik's and Oracle's JVM are different in the level of the architecture and instruction's set. Dalvik's is register-based and Oracles are stack-based. Generally Dalvik is designed to use fewer instructions than Oracle to achieve the same result making the step of interpretation more efficient. This security level, is possible thanks to the Dalvik VM design. Every time an application is running, a new instance of Dalvik VM is created so that each application can be run in an isolated thread. Expect .dex files there is one more file format that can be used and this is Optimized DEX .odex. Odex is the result of optimization that happens from Android system during application's installation. The optimisation has to do with the current device's specifications.

In case of NDK, it is not allowed to develop full applications that run outside of the Dalvik VM. It is used only to develop C/C++ libraries that are packaged inside the application's and can be called by the application within the VM using the previously mentioned JNI mechanism. From a security viewpoint, native development is very interesting due to unsafe characteristics os C/C++ (buffer overflows, race conditions etc).

Comparing desktop applications and android applications, android applications are quite different. Desktop applications are a collection of one executable file containing all the needed code and some independent resources like dynamically loaded libraries or other files. Android applications are a collection of independent prebuilt classes called components that can communicate with each other using a mechanism of message exchange. Specifically there are four different components which are the Activities, Services, BroadcastReceivers and ContentProviders. Lastly, all these resources and compiled code for distribution and installation purposes are included in a single file formatted as ".apk". The must important file is the AndroidManifest.

Some of the elements of an .apk file are listed below.

- AndroidManifest.xml
- *.so: shared object files
- classes.dex: compiled Java code of the application
- META-INF/: contains meta information about application's package
- lib/: contains platform dependent Native libraries
- res/: resource's files location
- assets/: includes raw images' location

After installation, installed files of each application can be found under one of the following locations inside a folder named after the application's package name.

- /data/app/: it is the normal installation location
- /system/app/: used for pre-installed system applications
- /data/asec/: used for secured applications
- /data/app-private/: used for third party protected applications

Also, during runtime some applications may need to use storage to store temporary generated files or a database to store data for later use. They use two locations, one internal and one external (SD card or emulated SD cards). These location are the following:

- /data/data/ (internal)
- /mnt/sdcard/Android/data/ (external)

In external storage, application can also store data in any location they want but in internal storage there is a API control mechanism. Also, the internal storage follows standard folder structure. Some of the must usually found folders listed are the following:

- lib/: Includes library files needed by the application
- files/: Includes files that is created by the developer and they are used by the application
- cache/: Includes files cached by the application
- databases/: Includes SQLite databases used by the specific application

Activities

Activities are responsible for UI composition and user interaction with the application. Each window of our application is a different activity. In more depth, activities are prebuilt classes that are used to draw UI content into the screen. These UI contents are called Views and some examples are buttons, labels and text input fields. In this category we can include also fragments. Fragments similar to Activities has to deal with UI composition but in smaller scale. They hold View objects but only such a part of a bigger Activity.

BroadcastReceivers

BroadcastReceivers are components that help our application to listen for broadcasted messages from other applications or the Android system itself. Broadcasts are special messages sent when an event occurs. Applications can register through the Android system to listen only for specific broadcasts. When a event occurs and a broadcast message is sent, the Android system automatically routes broadcasts to application that have registered to receive that specific broadcast. For example, during normal androids operation a message that has to do with incoming email could be sent.

ContentProviders

ContentProviders are components that manage access to central data repositories and help between applications data sharing. Using components like these the access to database from applications are easier as knowledge of lower level database management is not needed. For an example the "Contacts" database. All applications that want to access contacts like an email application does not need to use SQL queries to get the data. It just use the appropriate ContentProvider that manages "Contacts" data.

Services

Services are components that can run in the background without letting the mainś application activity to freeze waiting data. An example of such an component is an application that plays music files in the background. A service never has an UI, just process data. As mentioned earlier, services can also have an interface and using an Android Interface Definition Language (AIDL) to provide some functionality to other applications or services. There are three types of Services:

- Foreground Services: Services that are running in the background but interacts with the user by displaying a notification. For example a service an audio application that has to do with playing music.
- Background Services: Services that are running in the background without any interaction with the user.
- Bound Services: Special services that are connected with a component and can exchange data with it.

Resources

As described above an application for better interaction with the users need some visual UI elements like images or audio files. These kinds of additional files and static elements all called "Resources".

AndroidManifest

As we mentioned above AndroidManifest is one of the must important files in an android application as it allows Android system to know details of the application. It is in XML format declares setting about application's name, which of the Activities will be the main one and will be showed up first when the application is launched, in which components the application haw access, which services is been used by application, special permissions like if it is allowed to access the camera or microphone, which Android OS versions are supported by the application etc.

Component Activation

An android application is a collection of components and resources interconnecting all together as manifest file describes. Each of these components can be activated by sending special messages to it. For example, when a user opens the "Contacts" application and interacts with it and chooses one of the available contacts a list of options pop ups like using the contact's mobile phone number to call or the e-mail address in order to send an email. Depending on the user's choose a different independent Android application will be launched (the default e-mail client or a Dialer). To be possible this independence between applications android system uses **Intents**. The Intents are an activation mechanism through message passing and are used when an application's component has data to process and needs an other component to handle them.

There are two types of Intents:

- **Explicit Intents**: These Intents specifies exactly the class name of the component to handle the data. Also, it is possible to call component of an other application. In this scenario the full package name is needed.
- **Implicit Intents**: This Intents are more generic that Explicit. They are not targeting a specific component. By using Intent filters they construct more general actions with data to be performed, and all the available components that could use to handle them are triggered.

As shown in Figure 2.2, in order a component to be activated, an Intent with some data must be created and be passed to Android system. The Android system the desired component to handle the included data. Also intents can be used inside of the application. Most applications have more that one Activity so with the help of this mechanism Intents are used to activate other Activities and move on with the data. When you use an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the intent filters declared in the manifest file of other apps on the device. If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

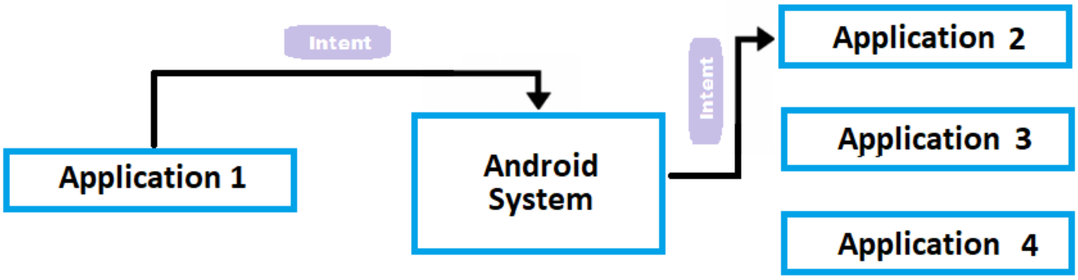


Figure 2.2: Intent workflow.

2.3 Libraries

A library is a collection of code that can be used by programmers. They are used from Android applications and system. Android developers can use two types of libraries, native C/C++ libraries and java (bytecode) libraries. Also native libraries can be categorized in two more categories, static and dynamic. The details of the mentioned categories are stated below.

Native Libraries

Native libraries are libraries that contain code that has been compiled using C or C++ and is working only in specific hardware CPUs. In Android development in which applications are programmed in Java, developers can use native libraries that is written in C or C++ by using the JNI framework.

Thanks to the reusability offered by already third-party implemented functionalities the use of prebuilt native libraries is very common. A very big disadvantage of using third-party libraries is that the developers are rarely unaware of new vulnerabilities fixes and updates of them making Android applications that uses them also vulnerable. As we can see at [14] in 80 studied programs the researchers found around 1,781 vulnerabilities and the most amazing was that 61 of the applications remained vulnerable without fixes for more than 5 years.

When these libraries that created as combination of object files compiled all together to form a single file are called **static libraries**. When an executable application uses statics libraries, all of them during compilation are copied into the application. Using static libraries all applications external dependencies are removed and the process of linking is faster since all the native code with the form of objects is compiled into the executable. A disadvantage of this technique is that updates to libraries need recompilation of the application in order that changes be included to our application.

When these libraries are loading at runtime from the applications when is needed during startup are called **dynamic libraries**. Contrary to static libraries, the code of dynamic libraries is not copied into application code. The application in order to use library's functions and code just includes the library's name in the binary file. With this technique the library is completely separate from application's source code and is more efficient. Applications executable is smaller in size. All these advantages of separate code has also some drawbacks. Using dynamic libraries the linking process is done during runtime of application instead of compile time, so it needs more time to load libraries and call functions. One more advantage is the the fact that the same dynamic libraries can be shared among different applications reducing by far the total needed memory. As far as, updates and changes in these libraries code does not require recompilation of them and final's application code.

Java Native Interface

Java Native Interface (JNI) is a framework that helps java that runs on top of Java Virtual Machine (JVM) to work with libraries coded in different native programming languages like C/C++. As Java is a portable language in order the developed applications that use JNI keep be portable must use compiled libraries for all possible CPU architectures. JNI as shown in Figure 2.3 helps calls made by Java code to be passed over the JVM to native implemented libraries and then possible returned data passed again back to the Java code in JVM. JNI must be thinking like an adapter that provides a mapping, through JNI.h header file, between Java and C/C++ function calling.

2.4 Android Security

For effective security, Android system designed with a multilayered security architecture using different mechanisms. These mechanisms target at integrity, confidentiality and availability of user's data, applications and devices. The main concept of this architecture is the android device to be safe even using the default configurations.

Android system can be separated in two spaces, kernel-space and user-space. In

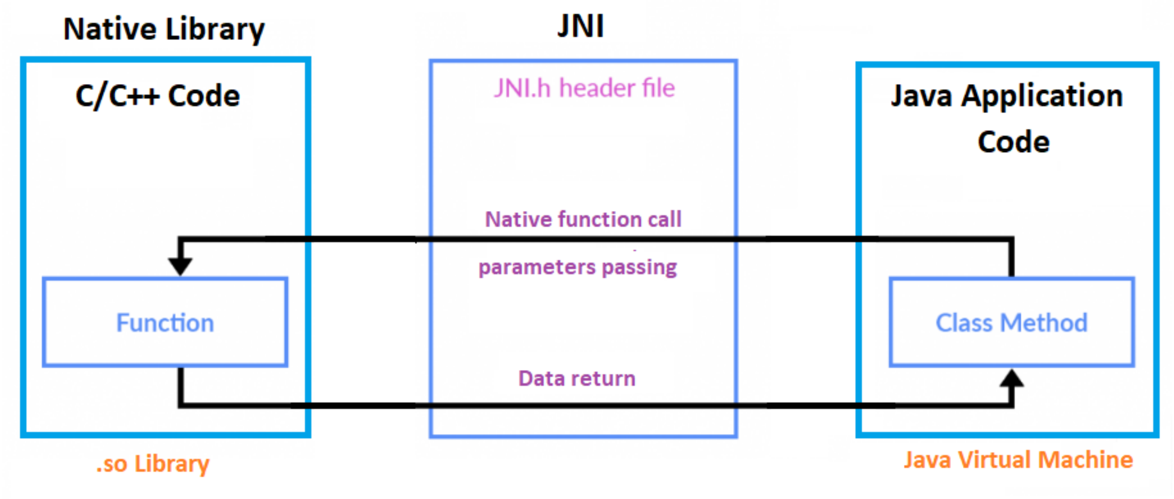


Figure 2.3: JNI architecture.

kernel-space code is allowed to perform low level operations accessing all hardware resources and memory. In user-space code is allowed to access only it own memory and hardware resources when the appropriate permissions are given by the users. The Android operating system utilizes two separate, but cooperating, permissions models.

2.4.1 Device Security

At device level, we have the following:

- **Locked Bootloader** and **OEM signature checking**, prohibiting new firmware installation from users into the devices.
- **Secure booting**, by checking firmware's integrity during boot time. During boot time, each next step before being executed is verified in terms of integrity and authenticity using strong cryptographic techniques. After Android 7.0 version devices that not pass the verification process is considered compromised and can not boot.
- **Partition locking** and **read-only storage mounts**, prohibiting changes to system's partitions.
- **File system and storage encryption**, providing limited access to user's data without running as the specific user. Full disk encryption is capable of using a single key encrypted and protected with the help of user's password and file based encryption offering encryption of different files using different keys.
- **Password based login** protection preventing not authorized users access device's data.

2.4.2 Application Security

At application level, thanks to Manifest file created each time by the developers the following extra security mechanism can be used by the Android system:

- **Digital signature validation**, during application's installation the Android system checks the digital signature of the .apk file in order to validate developer's identify. The developer's digital signature is created by the developer with out the needed of a central certificate authority (CA) like other certification protocols and is valid for 25 years. This is called the same origin policy and it is important for establishing a trust relationship among applications and updates signed by the same developer with the same signature. This is why it is important not to sign applications with publicly available certifications. System applications and services are signed by a number of platform keys so that the system components can run inside the same process and share the same resources.

- **Permission checking**, only if application signature validated successfully, Android through Manifest file checks in which devices resources and components application asks for access. This information is passed to users for the final approval by displaying a allow/deny dialog. The user's judgment is very important. For example, it would be very suspicious for a movie player application to need access to make phone calls. Then, if the user granted all permissions and the application can now be installed on the device.

Permissions

Permissions are a mechanism that helps to the user's data protection. Each application through manifest file declares needed permissions for **restricted actions** accessing devices resources like Bluetooth and camera or accessing **restricted data** like system's state and user's personal information. All that permissions are declared in the manifest file. Usually the developers are using pre-existing permissions defined in the android system but is also possible for developers to define custom permissions for restricting access to application's resources.

There are three different types of permissions:

- **Install-time permissions**: Permissions that are given to the application only during installation. It includes limited access to resources and user's information that can be characterized by the minimum affect to system. This permissions can be viewed only in application's details in App Store.

- **Normal Permissions**: Permissions that do allow access to user's private data and resources with minimum risk.

- **Signature Permissions**: Permissions are granted only to the apps that are signed by the same certificate.

- **Run time Permissions**: Permissions also known as **Dangerous Permissions** that gives to the application access to additional private user's data or resources and information. In these resources are included the protected APIs like camera, GPS, or Network functions and Cost related APIs like telephony or SMS/MMS functions.

2.4.3 Kernel Security

In the core of the Android system is the Linux kernel. The kernel is responsible for implementing many low level security mechanisms, including:

- Permission management.
- Process sandboxing (isolation).
- A secure Inter Process Communication (IPS) mechanism. Part of IPS are the Intents, Services, ContentProvides which mentioned above and Binders. Binders are high performance but high performance mechanism to call remote procedures.
- Read-only access to partitions that contains Android kernel, system libraries and application runtime.
- Use of Security-Enhanced Linux (SELinux) for in forcing access control policies and mandatory access control (mac) for processes. SELinux operates on the principle of default denial.
- Limited use of root permissions to core applications as with these permissions a malicious application could have access to all data and could modify the OS.

Application isolation (sandoxing)

The most important mechanism of the above is process isolation. With this mechanism Android system prevents from users access other users' resources and applications access others applications' memory, data or permissions protecting in this way from malicious applications.

To be accomplished such an isolation, following UNIX's approaches in each application and everything above the kernel Layer, including native libraries, is assigned during first installation into the device a **unique Linux user ID (UID)** also known as app ID and a **group ID (GID)** and runs in a different process at Dalvik VM. These IDs are not applicable only to application's processes but also to these directories that described above, that are created for each application in order to store data. In this way the applications are considered to be isolated, on two different levels, at the process level by running in a unique process and at the file level by having it's own private data folder.

UIDs

Except normal user application there are some services and applications that are running under specific UIDs and fewer running with traditional Unix's root permissions. Root services are using UID 0, the rest system services are using UIDs in range 1000 to 9999. These are defined in android_filesystem_config.h header file. For example, Bluetooth subsystem uses UID 1002 (AID_BLUETOOTH) and wifi subsystem 1010 (AID_WIFI). UID 1000 (AID_SYSTEM) is the main system server user, with special privileges. All applications UIDs are generated automatically by the system and are starting from 10000 (AID_APP), in the of app_XXX or uY_aXXX if the Android versions supports multiple users login, where XXX is an incremented number (offset) from AID_APP number and Y the specific Android user's ID.

At this point we must refer to an exception with the unique characteristic. A developer can use the same certification to more than one application in order to share on purpose the same processes, memory or data and as a result to use the same UID. This UID is called **shared user ID**. This is vary common practise when there are offered two different versions of the same application, a free and a paid. With this technique a user can go from the one to the other without losing any data. Also, shared user IDs are used a lot by system applications ans services, as for system is very important the ability to use the same resources across different services.

GIDs

The GID depends on the particular permissions the application asked for. For example, if the application needs to access the device's camera must be member of GID "camera". Also can be a member of more than one groups at the same time. For example member of GID "inet" for internet access and "net_bt" for Bluetooth access. The permissions for different groups are located in the platform.xml file in the /system/etc/permissions/ directory on our device.

Chapter 3

Fuzzing

There are about three basic methods for detecting security flaws in software:

- static program analysis
- static program analysis
- fuzzing.

Static program analysis is performed using tools that automatically inspect code and find code with potential faults using pattern matching. These tools are a useful first line of protection against security vulnerabilities since they are quick and can uncover a lot of small issues, but they are also prone to reporting a lot of false alarms. Static analysis is a type of analysis that is performed on source code without the need to run it.

Manual code inspection is the process of inspecting code before it is released. It is used in almost all software development processes and is capable of detecting major errors. During the inspection, security professionals look over the code, paying special attention to security flaws in the code. It's an approach very slow but it can be used to any product without requiring a lot of tools, and it can disclose design problems and coding issues that automated techniques can't detect.

Fuzzing is the most used method for detecting software security flaws. Fuzzing is the process of repeatedly running a program with various input variations in order to find security flaws by causing crashes. These input variation called test cases. Test automation, or the ability to run tests automatically, is required for fuzzing. Each test must also execute quickly, and after each test the applications state must be reseted.

Among the three approaches as we can see and from Table 3.1, fuzzing seem to be the most suitable for his speed end efficiency. In the following paragraphs we will go through this well-known technique and its history in order to better understand why it is so crucial for software testing.

In this section we provide a viewpoint on fuzzing, including basic approaches, background information, and obstacles in enhancing fuzzing.

Technique	Difficulty	Accuracy	Stability
Static analysis	Easy	Low	Good
Manual inspection	Hard	Low	Bad
Fuzzing	Easy	High	Good

Table 3.1: Comparison of different techniques.

3.1 History

Back in 1988, Professor Barton Miller proposed a new technique to test the reliability of command line programs. According to Miller's et al. initial study [15] the term "fuzz generator" is used to refer to a program that generates a stream of random characters to be consumed by a target program. Then, after inserting these random data into the target program each of the found crashes is being debugged in order to determine the cause and categorized each of the detected failure. According to the same study a significant number of tested programs (between 24% and 33%) crashed or hanged on each system.

Three further studies were conducted after that. Five years later, in 1995, one [16] re-testing UNIX command line utilities increasing the number of utilities and UNIX versions tested, and also extending testing to X-Window GUI applications, the X-Window server, network services, and the standard library interface. Of the commercial systems that tested, 15-43% of the command line utilities crashed, 6% of the open-source GNU utilities and 9% of the utilities distributed with Linux. The most incredible part is that the reasons of these crashes were nearly identical to those found in the 1990 study. Of the X-Window applications that tested, 26% crashed or hang due to random valid keyboard and mouse events. The second study, done in 2000 [17] targeting Microsoft Windows platform found that the percentage of crashes scaled up to 45%. In this study the authors used the Win32 interface, to send random valid mouse and keyboard events to the application programs. Finally, in 2006, a third study [14] targeting Apple's Mac OS X revealed that 7% of command line utilities crashed. Very interesting is the fact that 73% of GUI-based utilities that tested crashed on random valid mouse and keyboard input.

Many of the problems discovered in the 1990s UNIX study were still present in 1995 and 2006, despite 16 years of published studies, increased use of fuzz testing, and freely available fuzz tools. As a result of these excellent results, fuzzing became one of the most promising techniques for vulnerability detection, and now it plays an essential part in software testing.

3.2 How Fuzzing Works

The fuzzing process includes the following basic phases:

- Identification of target
- Identification of inputs
- Generation of fuzzed data
- Execution of fuzzed data

- Monitoring for exceptions
- Reporting

Target identification is the first and most important phase of vulnerability discovery. During this stage, decisions are made not only about the targeted program, but also about testing specific functions and libraries within our target. Knowing your actual target, such as which company developed the specific software, history for previous discovered vulnerabilities and possible frameworks and libraries used are important part of this phase. For example, if a framework was used to sanitize input data, it would be waste of time for our fuzzer to trying mutation of specific types of inputs.

Identification of all possible inputs is vital for a successful fuzzing. The system under test is tested through its interfaces. All these inputs are also known as attack surface. The majority of exploitations are caused because applications accepting and processing user input without sanitizing and verifying it. It's critical to identify the attack surface, or all external interfaces through which an attacker could send potentially dangerous data. All types of parsers for protocol and file handling or processing, as well as application programming interfaces (API) and dynamic injection of external libraries, are included in the attack surface. For network based system attack surface might vary from low level packets such as Ethernet packets, up to the TCP/IP stack. It's critical to figure out which protocols and formats the system can handle.

After identification of inputs, **generation of fuzzed data** takes place and new test cases are generated. This data may be some predefined values, mutation of existing real input data or some dynamically generated data depending upon the target and inputs. The main techniques that used during that phase are mutation and generation.

The phase of execution of fuzzed data involves **launching a target process** and then send mutated data packet towards the target. Execution happens along with the generation. In its iteration new data are generated and are inserted into the target.

Monitoring for exceptions is one of the most vital processes of fuzzing. During fuzzing, we are sending thousands of fuzzed data into the target and ultimately causing that target to crash will be a futile exercise, if we are not able to find which data were responsible for crash. It's critical to be able to duplicate crashes once they've been discovered.

Finally, **reporting** the findings of the fuzzing process in the form of a report, it is important as it helps in an overview of all process.

3.3 Types Of Fuzzers

Depending on how much information we know about the target system fuzzing techniques are classified as black box, white box, or gray box.

In **white-box fuzzing** source code is known to the analyst. The understanding of the target program's core logic is mandatory in order to uncover all possible execution pathways in the target program. These fuzzers often use dynamic symbolic execution to construct test cases or locate all potential inputs, which are then used by black-box or grey-box fuzzers. White-box techniques are the only way to get 100% coverage.

In **black-box fuzzing**, source code is not available and testing exercises a target program or process without examining any code, whether source code nor binary/assembly code. This technique can observe only the input/output behavior of the target and according to the output it generates more data for input. Black-box techniques have a lot of shortcomings. For example, if a fuzzer is trying to execute the "then" branch of the conditional statement "if (x < 3) then", with x a 32-bit value, would have to supply 2^{32} different values. This is why random testing usually yields at low code coverage [18]. More effective for the last example would have been to send boundary values like 0x0000000 and 0xfffffff. This approach will be discussed later. Due to the majority of developed software is delivered closed source, black-box fuzzing is the most used method now days.

In **grey-box fuzzing** access to the source code is not necessary. It extends black-box fuzzing by using white-box fuzzing techniques. Testing exercises a target program or process by instrumenting binary/assembly code using a debugger and monitoring various statistics.

The production of new test cases is one of the most significant components of fuzzing. The following two ways are commonly used to produce these inputs or test cases:

Mutation-based test cases are created by altering legitimate, known-good samples, such as valid files or network traffic recorded. Mutations include bit or byte flips, setting a specific byte to zero or the maximum value, and other simple operations. This increases the likelihood that the input is still valid, but it may cause unexpected behavior in the target system. The user must give the fuzzer a seed to generate the first test case. As can be seen, the mutation-based fuzzer does not require knowledge of input data specifications or protocol usage because mutations affect the original seed without evaluating whether it corresponds with a specified syntax. This method has the advantage of requiring little or no target knowledge. All that is required are some high-quality data samples.

In **generation-based** test cases the user needs to provide a model for the protocol under test, either by writing it from scratch, or by reusing an existing protocol model. In second option, a set of specifications used on the target system inputs to generate new fuzz test data. In order to use generation-based fuzzing, it is necessary to know the syntax of the targets inputs, including their format and underlying protocol. Generation-based fuzzers are a grate option of use when the target program uses techniques for checking data's validation. Generating new test cases from complex or unknown protocols structures is a process too challenging. Studying the test specification and creating the test cases from scratch takes a large amount of time but the extra information obtained through understanding the format leads to higher-quality test cases.

According to research [19] the authors suggests that generation-based fuzzers possess a higher effectiveness in finding vulnerabilities. The difference of the two is being showed in Table 3.2.

Technique	Difficulty	Priori knowledge	Coverage	Ability to pass validation
Mutation based	Easy	Not Need	Good	Weak
Generation based	Hard	Needed	High	Strong

Table 3.2: Comparison between generation based and mutation based fuzzers.

Two more categories that fuzzers can be classified based on the technique of exploring programs paths are: directed fuzzing and coverage-based fuzzing.

On one hand, a **directed fuzzer** aims at generation of test cases that cover target code and target paths of programs trying to achieve faster tests on programs. A fuzzer like that spends most of its time on reaching specific target locations without wasting resources stressing unrelated program components like discovering new code branches. The majority of directed fuzzers are based on symbolic execution. With symbolic execution which is a white-box fuzzing technique that program analysis and constraint solving to synthesize of inputs is used for discovering new program paths.

On the other hand, the goal of a **coverage-based fuzzer** is to generate test cases that cover as much code as possible and find as many errors as possible. The coverage-based fuzzing method as shown in Figure 3.1, using a feedback mechanism is extensively employed by modern fuzzers and has proven to be extremely effective and efficient. It starts with a set of user-specified input seeds; if none are available, the fuzzer will create one on its own. The fuzzer then changes these initial seeds and runs again the program under test with the new inputs. If during the last execution the fuzzer found new control-flow edges or code branches, the input is considered as interesting and kept as a seed for further mutation; otherwise, it is discarded. Coverage information is typically monitored using lightweight program instrumentation, which does not slow down the program's execution pace. This basic technique has been shown to be quite useful in locating flaws in real-world programs.

According to whether there is feedback between the monitoring of program execution state and the development of test cases, fuzzers can be categorized as dumb fuzz or smart fuzz. **Smart fuzzers** can adjust the generation of new test cases according to information that show how much latest test cases affect the program's behavior. For mutation based fuzzers, feedback information could be used to determine which part of test cases should be mutated and the way to mutate them. Smart fuzzers generate better test cases and improve efficiency, while dumb fuzzers improve testing speed.

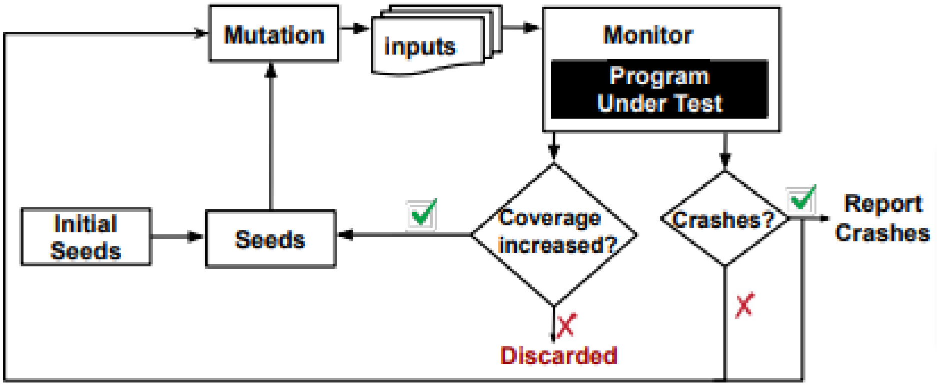


Figure 3.1: Coverage-based fuzzing process. [3]

3.4 Metrics

One of the major challenges of fuzzers is measuring their effectiveness. The simplicity of crash detection has led to the widespread use of crash count as a performance metric for comparing fuzzers. However, crash counts have been shown to yield inflated results, even when combined with de-duplication methods as coverage profiles and stack hashes. Fuzz test tools are frequently used in conjunction with a de-duplication mechanism, which determines whether a detected error is identical to, or similar to a previously detected fault and should be recognized as originated from the same root cause. Also, obtaining 100% code coverage doesn't necessarily mean all bugs have been found, it's certainly true that no bugs will be found in code that hasn't even been executed. In a study conducted by Yuwei Li et al. [20] are proposing a set of metrics, which can be classified into six categories: quantity of unique bugs, quality of bugs, speed of finding bugs, coverage and overhead.

Quantity of Unique Bugs is one of the most important metrics as there exists randomness with a fuzzing process the same bug in the same position on code can be measure more than once time. In this case it would be wrong to include all those in the report.

Quality of Bugs, while finding bugs we should also examine the severity of bugs and the effectiveness of fuzzers in finding rare bugs. A fuzzer which can find more high quality bugs should be considered as better. Speed of Finding Bugs it's critical to find defects quickly and efficiently, especially when the time budget is limited. A safe measurement of the speed of finding bugs is to measure the time that it took to find only unique bugs.

Code Coverage metrics are used to measure a fuzzer's capability of exploring paths. Code coverage criteria, often known as test coverage criteria, are an important part of software testing research. They have been studied [21] [22] [23], and are notably used to evaluate the effectiveness of test suites to exercise a piece of software. Different granularity levels, such as function, basic block, edge, and line coverage, can be used to measure coverage metrics. According to the same study [20], the most common coverage criteria are the follow.

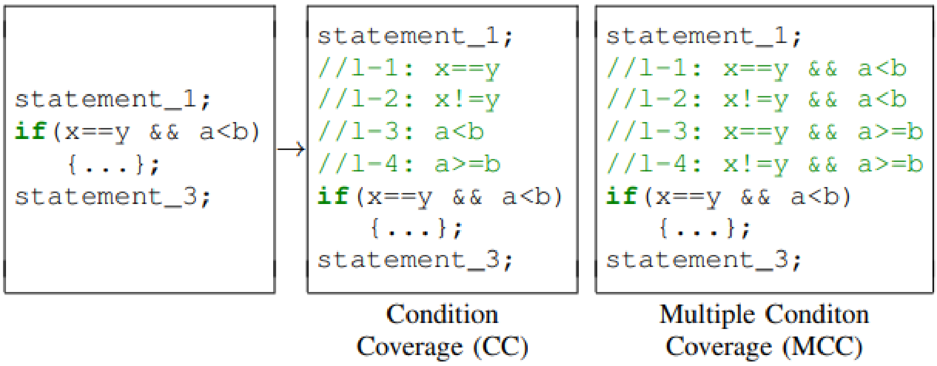
Control-flow and call graph criteria:

- **Statement Coverage (SC)**: requires the statements of the target program to be reached.
- **Decision Coverage (DC)**: requires to be activated both the true and false path of each in a decision point. This is equivalent to covering all the edges in the control-flow graph of the program.
- **Function Coverage (FC)**: requires all functions entry-points to be reached.

Logic expressions criteria:

- **Condition Coverage (CC)**: requires a fuzzer to activate both true and false values for each of the conditions in any target program's decision point. With atomic conditions we refer to logical expressions that cannot be divided into other simpler expressions.
- **Decision Condition Coverage (DCC)**: requires a fuzzer to satisfy both DC and CC.
- **Multiple Condition Coverage (MCC)**: requires a fuzzer to activate all the combinations of truth values.

In Figure 3.2, we can see an explanation of how two different criteria are used by the fuzzer. With green labels are shown the conditions that must be fulfilled to be measured as code covered.



```
statement_1;
if(x==y && a<b)
   {...};
statement_3;
```

```
statement_1;
//1-1: x==y
//1-2: x!=y
//1-3: a<b
//1-4: a>=b
if(x==y && a<b)
   {...};
statement_3;
```

```
statement_1;
//1-1: x==y && a<b
//1-2: x!=y && a<b
//1-3: x==y && a>=b
//1-4: x!=y && a>=b
if(x==y && a<b)
   {...};
statement_3;
```

Figure 3.2: Coding Coverage Criteria. [3]

Overhead metrics measures how many computing resources a fuzzer needs during the fuzzing process. The overhead of a fuzzer can be measured by the following concrete metrics: CPU utilization, memory consumption and the amount of disk read/write.

Chapter 4

Targeting Android

The most important part of fuzzing as described in the previous chapter is to identify the possible attack surface. So, in this chapter we will present all the appropriate tools and techniques in order to reverse engineer and look inside a third party Android application for which we don't have the original source code and understand its different components. Analyzing the source code we can understand the application's structure and find different kind of inputs. Next we will see how we can instrument an in memory fuzzer using Frida framework [5] in order to send generated data to these inputs trying to cause crashes and find possible vulnerabilities. Using Frida we can develop a hook and hijack any Java method and class and inject custom code. Working at a such higher level we can even fuzz native calls to .so libraries without needing a lower level fuzzer.

The importance of such and approach can be seen by the fact that at the time of writing this thesis all available tools for Android application fuzzing such as radasma-android [24], Honggfuzz [25], melkor-android [26], MFFA [27], Afl++ [28], libFuzzer [29] and other are all designed to fuzz only binary compiled applications and libraries and even worst the majority of them are working as white-box fuzzers.

The main goal of fuzzing is to find crashes through logic flaws in applications implementation when developed with Java or identify illegal memory operations in native libraries developed with type-unsafe languages such as C/C++. In most cases due to the way the Android system is developed the crashes we be a combination of the above. The illegal memory operations include:

- **Buffer Overflows**, occurred when a program writes more data to buffer that it's size. Usually, it is unknown what was overwritten, and this often times leads to undefined and unpredictable behavior for programm execution.

- **Null Pointer Dereferences**, that happens when a program does not check for a NULL pointer and attempts to use that pointer as if it was normal variable.

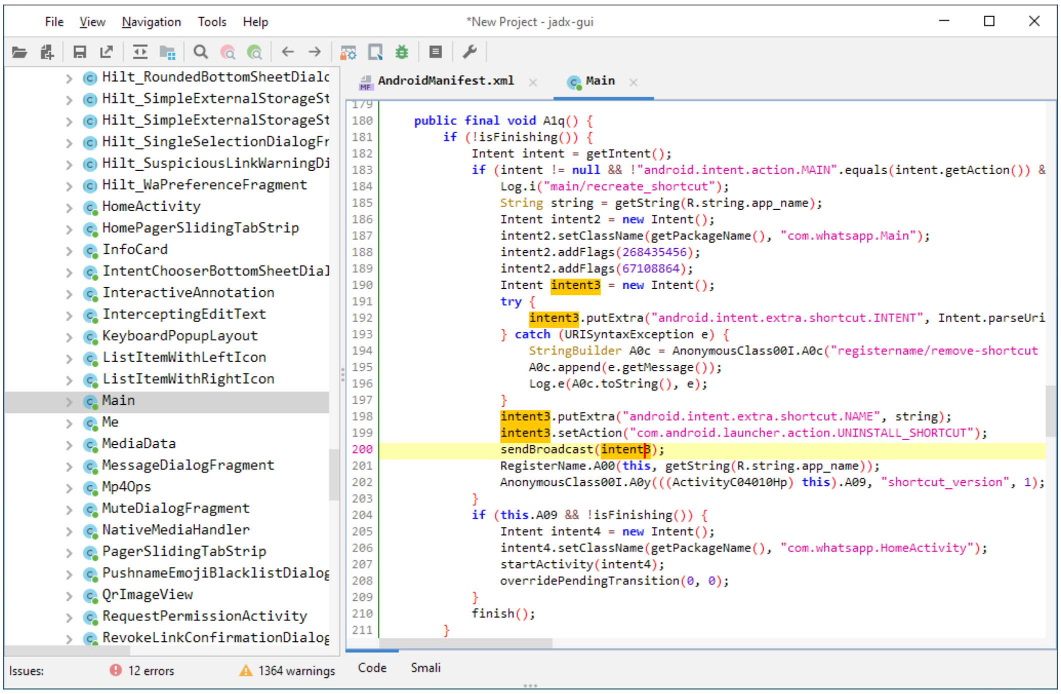
- **Double Memory Frees**, that happens when a dynamically-allocated memory regions is tried to be freed twice. The first time is valid and the program will continue its normal execution, but in the second time will cause corruption in the memory allocator's internal data structures, causing a segmentation violation, abort, or other undefined behavior.

- **Use-After-Frees**, that occurs when dynamically-allocated memory is freed and then at some other point of execution the program tries to accessed it. The cause of the problem is that after memory free operation this memory no longer belongs to the program, and but contains valid data. Such scenarios often cause undefined behavior or crash.
- **Uninitialized Memory Accesses**, that happens when during a program's execution, there are not proper memory allocation and maybe regions of its stack and heap memory may be will going to be reused containing already data that should be properly be reset.

4.1 Tools

Jdax

Jdax [4] is a command line and GUI **decompiler** tool. This tool is used to decompile Android Dalvik bytecode from dex, aar, aab or zip files files into readable Java source code. Also it can be used for AndroidManifest.xml file or other resources decoding and extraction from resources.arsc. An example of jdax-gui can be seen in Figure 4.1.



```
    public final void A1q() {
        if (!isFinishing()) {
            Intent intent = getIntent();
            if (intent != null && !"android.intent.action.MAIN".equals(intent.getAction()) &
                Log.i("main/recreate_shortcut");
                String string = getString(R.string.app_name);
                Intent intent2 = new Intent();
                intent2.setClassName(getPackageName(), "com.whatsapp.Main");
                intent2.addFlags(268435456);
                intent2.addFlags(67108864);
                Intent intent3 = new Intent();
                try {
                    intent3.putExtra("android.intent.extra.shortcut.INTENT", Intent.parseUri
                } catch (URISyntaxException e) {
                    StringBuilder A0c = AnonymousClass00I.A0c("registername/remove-shortcut
                    A0c.append(e.getMessage());
                    Log.e(A0c.toString(), e);
                }
                intent3.putExtra("android.intent.extra.shortcut.NAME", string);
                intent3.setAction("com.android.launcher.action.UNINSTALL_SHORTCUT");
                sendBroadcast(intent3);
                RegisterName.A00(this, getString(R.string.app_name));
                AnonymousClass00I.A0y(((ActivityC04010Hp) this).A09, "shortcut_version", 1);
            }
            if (this.A09 && !isFinishing()) {
                Intent intent4 = new Intent();
                intent4.setClassName(getPackageName(), "com.whatsapp.HomeActivity");
                startActivity(intent4);
                overridePendingTransition(0, 0);
            }
            finish();
        }
```

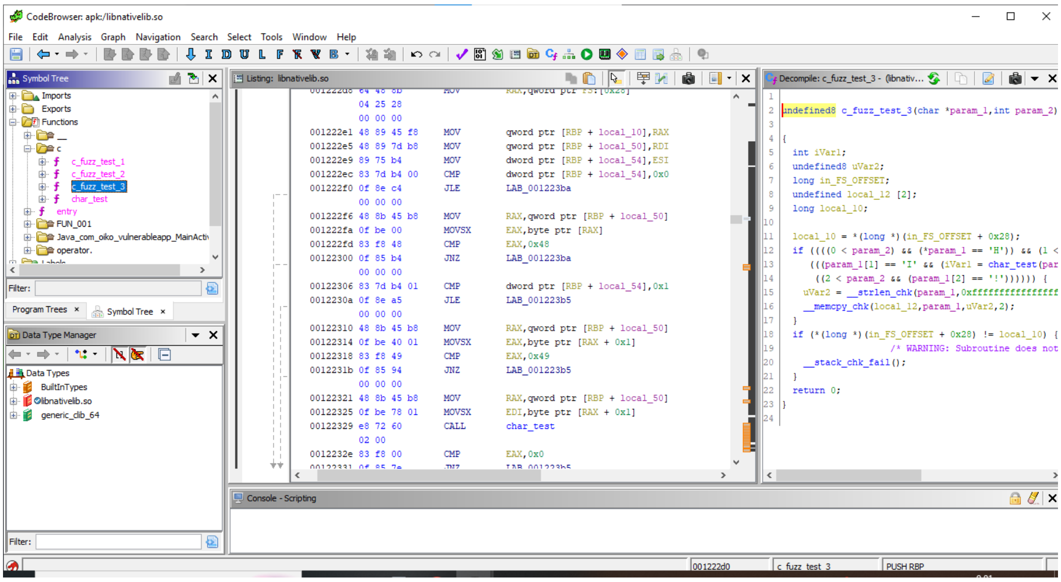
Figure 4.1: Jdax interface example. [4]

Apktool

Apktool [30] is a tool used for **reverse engineering** 3rd party and closed source Android applications. It can decode all the included resources capable for modification and rebuild them to a working apk file ready for installation to devices. The main difference from tools like jadx is that it decompiles Android bytecode to smali code.

Ghidra

Ghidra [31] is a Java based software reverse engineering (SRE) framework created and maintained by the National Security Agency Research Directorate. It includes analysis tools and features for disassembling, assembling, decompilation, graphing, and scripting for native compiled code. An example of Ghidra gui can be seen in Figure 4.2.



```
          04 25 28
          00 00 00
001222e1 48 89 45 f8    MOV      qword ptr [RBP + local_10],RAX
001222e5 48 89 7d b8    MOV      qword ptr [RBP + local_50],RDI
001222e9 89 75 b4       MOV      dword ptr [RBP + local_54],ESI
001222ec 83 7d b4 00    CMP      dword ptr [RBP + local_54],0x0
001222f0 0f 8e c4       JLE      LAB_001223ba
          00 00 00
001222f6 48 8b 45 b8    MOV      RAX,qword ptr [RBP + local_50]
001222fa 0f be 00       MOVSX    EAX,byte ptr [RAX]
001222fd 83 f8 48       CMP      EAX,0x48
00122300 0f 85 b4       JNZ      LAB_001223ba
          00 00 00
00122306 83 7d b4 01    CMP      dword ptr [RBP + local_54],0x1
0012230a 0f 8e a5       JLE      LAB_001223b5
          00 00 00
00122310 48 8b 45 b8    MOV      RAX,qword ptr [RBP + local_50]
00122314 0f be 40 01    MOVSX    EAX,byte ptr [RAX + 0x1]
00122318 83 f8 49       CMP      EAX,0x49
0012231b 0f 85 94       JNZ      LAB_001223b5
          00 00 00
00122321 48 8b 45 b8    MOV      RAX,qword ptr [RBP + local_50]
00122325 0f be 78 01    MOVSX    EDI,byte ptr [RAX + 0x1]
00122329 e8 72 60       CALL     char_test
          02 00
0012232e 83 f8 00       CMP      EAX,0x0
```

```

undefined8 c_fuzz_test_3(char *param_1,int param_2)

{
  int iVar1;
  undefined8 uVar2;
  long in_FS_OFFSET;
  undefined local_12 [2];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  if ((((0 < param_2) && (*param_1 == 'H')) && (1 <
     (((param_1[1] == 'I' && (iVar1 = char_test(par
      ((2 < param_2 && (param_1[2] == '!')))))) {
    uVar2 = __strlen_chk(param_1,0xffffffffffffffff
    __memcpy_chk(local_12,param_1,uVar2,2);
  }
  if (*(long *)(in_FS_OFFSET + 0x28) != local_10) {
                    /* WARNING: Subroutine does not
    __stack_chk_fail();
  }
  return 0;
}

```

Figure 4.2: Ghidra interface example.

Frida

Frida is a **Dynamic instrumentation toolkit** developed by Ole Andre V. Ravnas and sponsored by NowSecure letting the injection of JavaScript code (injection mode) or libraries (gadget mode) into native applications supporting Windows, macOS, GNU/Linux, iOS and Android. This injecting code at runtime of an applications and in our case Android application lets as obtain behavioral information during its execution and modify the normal application's flow by calling method's parameters and returns as these code is executed with full access to memory, hooking functions and even calling native functions inside the process. The previous technique of dynamic modification is also known as the instrumentation process. All the instrumentation code is developed in JavaScript or TypeScript. Also, Frida in order to provide a higher level of orchestration except the instrumentation code that is running in targeted application provides in different languages like Python, Java, Swift etc. bindings through APIs so that we can automate more all the process.

Thanks to instrumentation process we are not only allowed to modify the execution flow of the given application but also to modify registers and give certain instruction patterns making Frida the perfect tool for:

- Reverse engineering, allowing to quickly obtain information about a binaries and processes when code is not available and static analysis time is not possible or there is a strict timelines.
- Fuzzing, controlling data throughout the execution of the application and method callings trying to cause application's crashes or race conditions leading to vulnerabilities.
- Malware Analysis, by the observation of abnormal application's flow and suspicious APIs usage.
- Taint Analysis, by checking all that variables that altered by the user and taking advantage of this we can examine selected parts of memory regions and registers to see how they are affected by the user's behavior.
- Measure performance, of specific code blocks by injecting dynamically code to gather statistics.

In Figure 4.3 we can see Frida's architecture. **Frida-agent** containing the instrumentation javascript file is injected into target applications. **Frida-core** contains the main injection code through which you can inject code into a process, creating a thread running QuickJS that runs our instrumentation JavaScript code. Also, can enumerate installed applications, other running processes or other connected devices. **Frida-gum** allows the operations like hooking and memory management using C functions. **GumJS** contains the JavaScript bindings and with the help of **Gum's APIs** we can hook functions, enumerate loaded libraries, their imported and exported functions, read and write memory or scan memory for patterns. Also, through **p2p-dbus** there is a bi-directional communication between the localhost application and the javascript code running inside the target process. Also using Frida, we can interact with the application so we can bypass many of the techniques used by the developers to secure the application. For example, we can bypass the login screen and authenticate without using a password gaining this way access to information that would not otherwise be available.

Thanks to frida's agent while Android ART loads the application's dex and .oat files in order to run the application a native shared library FridaDroid in charge of all dynamic instrumentation and our javascript is loaded in parallel. Next, FridaDroid hook ART functions depending in the javascript code that we provided to the application and modifies application's behavior accordingly.

Frida, is such a powerful framework that except of the following tools that are included provides an API so that we can develop our personal tools.

- Frida Cli, a tool aimed at rapid prototyping and easy debugging, for more Use Frida -h.
- frida-ps, a command-line tool for listing processes useful when interacting with a remote system.
- frida-trace, a tool for dynamically Monitoring/tracing of devices Method calls useful for debugging method calls on every event in mobile application.
- frida-discover, a tool for discovering internal functions in an application, which can be used later to be traced by using frida-trace.

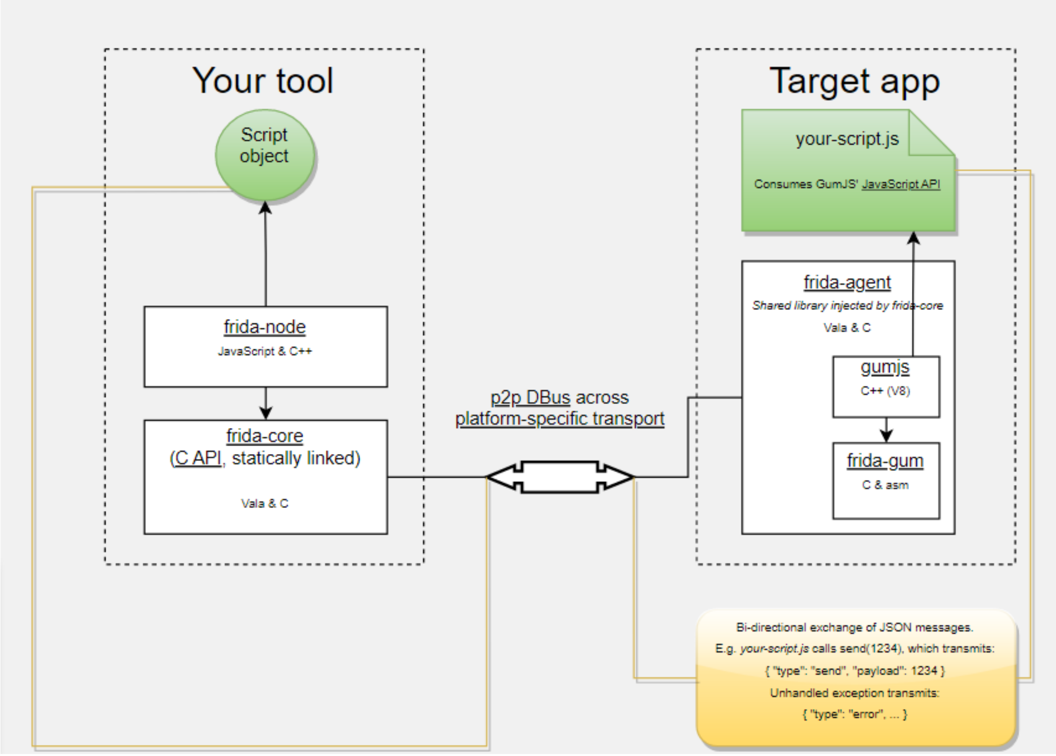

Your tool
Script object
frida-node
JavaScript & C++
frida-core
(C API, statically linked)
Vala & C
p2p DBus across platform-specific transport
Target app
your-script.js
Consumes GumJS' JavaScript API
frida-agent
Shared library injected by frida-core
Vala & C
gumjs
C++ (V8)
frida-gum
C & asm
Bi-directional exchange of JSON messages.
E.g. your-script.js calls send(1234), which transmits:
{ "type": "send", "payload": 1234 }
Unhandled exception transmits:
{ "type": "error", ... }

Figure 4.3: Frida Architecture. [5]

- frida-ls-devices, a command-line tool for listing attached devices useful when interacting with multiple devices.
- frida-kill, a command-line tool for killing processes.

4.2 My vulnerable app

In order to effectively check our fuzzer we built our personal vulnerable application. This application B.1 includes two main buttons, one that calls a JNI implemented function in order to update a label and one that calls Java code to update the same label with different text. Also, in the native library we inserted a lot of vulnerable code containing stack and buffer overflows and two different types of functions. Some function are used only by other native components of the library and the others only for being called by the Java code through JNI. In Listings B.1 and B.2 we can see our code.

4.3 Our Fuzzing Architecture

Our fuzzing approach for implementing our own **in memory, dynamically instrumented and coverage guided fuzzer** is to use part of the logic of one of the must effective fuzzers and transfer them in memory. This fuzzer is AFL [32]. The idea for this approach comes from the fact that AFL is one of the most efficient tools for that fuzzing but lacks abilities that can be offered by Frida toolkit. These abilities that are coming from dynamic instrumentation are things like bypassing login forms or other authentications mechanisms, attacking multi-language applications with the same implementations and other. Attacking multi-language applications is very crucial when attacking Android applications as they use in most cases a mix of Java and C/C++ code and normally would need to use a different approach for fuzzing java bytecodes and native libraries in same time.

Our fuzzer is working as follow:

- Step 1. **Initial test case loading**: All the test cases provided by the user will be loaded into a memory queue.
- Step 2. **Next test case trimming**: Next test case from the queue will be taken and the fuzzer will attempt to trim it to smallest size.
- Step 3. **Test case mutation**: The test case is going to be mutated by using a variety of strategies and inserted in the target system. These strategies will be analyzed later.
- Step 4. **Crash or new path discovery**: If the generated by mutation test case caused new state path discovery or cause an application crash will be added to the queue.
- Step 5. **Repeat**: Repeat from Step 2.

In Figure 4.4 we can see the main mutation strategy we will follow. In **Deterministic stage**, 6 deterministic types of mutation operators (bitflip, byteflip, arithmetic inc/dec, interesting values, auto extras, user extras) are chosen and used in order in each of the test cases. In **Havoc stage**, which is probabilistic is determined a number *R* of wanted test cases to be generated and next a number of operations are chosen (bitflip, byteflip, arithmetic inc/dec, interesting values, random byte, delete bytes, insert bytes, overwrite bytes) and applied in order to the initial test case in order to create these *R* test cases. For our fuzzer we chose $R = 1$. In **Splicing stage** as fuzzer fail to discover any unique crashes or new path uses cross over technique in order to generate new test cases from combination of all the initial test cases and feed them to Havoc stage. The last stage is rarely used. These three stages are also known as schedulers.

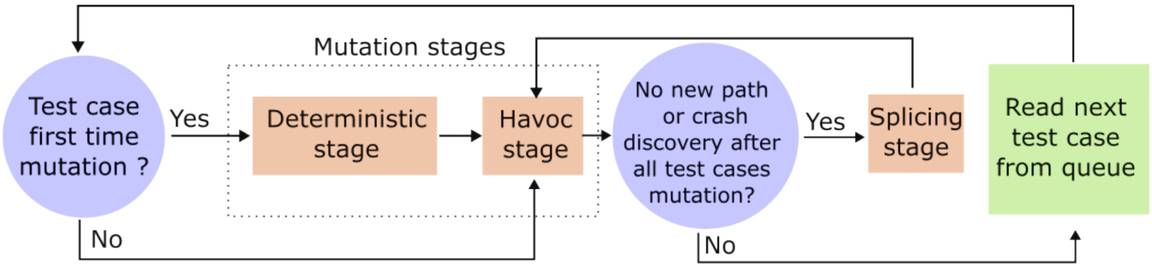

Mutation stages
Test case first time mutation ?
Yes
No
Deterministic stage
Havoc stage
No new path or crash discovery after all test cases mutation?
Yes
No
Splicing stage
Read next test case from queue

Figure 4.4: Mutation strategy.

The operations we mentioned above are the following:

- **bitflip:** Invert one or several consecutive bits in a test case with 1 bit stepover. Operation modes: bitflip 1/1, bitflip 2/1, bitflip 4/1.
- **byteflip:** Invert one or several consecutive bytes in a test case, with 8 bits stepover. Operation modes: bitflip 8/8, bitflip 16/8, bitflip 32/8.

- **arithmetic increase/decrease:** Perform addition and subtraction operations on one byte or more consecutive bytes. Operation modes: interest 8/8, interest 16/8, interest 32/8.
- **user extras:** Overwrite or insert bytes in the test cases with user-provided tokens. Operation modes: user (over), user (insert).
- **auto extras:** Overwrite bytes in the test cases with tokens recognized by the fuzzer during bitflip 1/1.
- **random bytes:** Randomly select one byte of the test case and set the byte to a random value.
- **delete bytes:** Randomly select several consecutive bytes and delete them.
- **insert bytes:** Randomly copy some bytes from a test case and insert them to another location in this test case.
- **overwrite bytes:** Randomly overwrite several consecutive bytes in a test case.
- **cross over:** Splice two parts from two different test cases to form a new test case.

The fuzzer is created with Frida's instrumentation language, javascript. Our script is more that 1k lines of code. At the start, Listing 4.1 we can see the place where we put the basic information about the fuzzing process. In this part we can set the targeted function name and information about the specific function, where it is located (target_module), the input parameters (target_function_arg_types) and what is returned (target_function_ret_type) in form of arrays. Also, we can specify the initial corpus that will be used by our fuzzer as first input and then as bases for the mutation stages. Next, we can see the Boolean variables deterministic_stage_enabled and havoc_stage_enabled where is defined if during the mutation will be used the specific stages.

```
///////////////////////////     TARGET CONFIG ///////////////////////////

var debugging_enabled               = false;
var target_module                   = 'nativelib.so';
var target_function                 = 'my_function';
var target_function_ret_type        = 'int';
var target_function_arg_types       = ['pointer', 'int'];
var watch_modules                   = [];

var corpus                          = ['AAAAA'];

var deterministic_stage_enabled = true;
var havoc_stage_enabled             = true;

                        [ ... code omitted ... ]
```

Listing 4.1: Target Configuration part.

In the next part of our code, Listing 4.2 fuzzer's user can put code that can dynamically instrument the java android part of our application. Java instrumentation is important as we can bypass authentication mechanisms or change the normal operation of java methods that could affect our fuzzer performance and effectiveness.

For example, this code changes the behavior of setText function which is called by button2 to set the text of the label.

```
                         [ ... code omitted ... ]

//////////////////////      JAVA INSTRUMENTATION   //////////////////////

Java.perform(function x() {
    var java_string = Java.use("java.lang.String");
    var main_activity_class = Java.use("com.oiko.vulnerableapp.mainactivity");

    // Hooking setText() function
    main_activity_class.setText.overload(string_class).implementation = function (x) {
        console.log("Original call: setText(" + x + ")");
        var new_string = java_string.$new(’Hello World’);

        this.setText(’Our modified setText Input’);
 };

                         [ ... code omitted ... ]
```

Listing 4.2: Java instrumentation code.

The next part is the fuzzer_Test_One_Input abstract method. This method is wrapper of the targeted function and called every time a new mutated test case has been generated. The input parameters are the generated mutated data and data length. This part is very important as the behavior of target functions can vary so the user must define how the fuzzer will call the function. For example, a function can have more than one targeted parameters or some additional values that the are needed to be set in order to run. A fuzzer could not guess all these peculiarities.

```
                         [ ... code omitted ... ]

/////////////////////      FUZZER INSTRUMENTATION  /////////////////////

function fuzzer_Test_One_Input(data, size) {

    // data: ArrayBuffer
    // size: int

    // char *      as parameter      : Memory.allocUtf8String("str")
    // c string    as return value   : Memory.readCString(arg)

    data = Memory.allocUtf8String(uint8array_to_str(data));

    var ret = native_function_handler(data, size);
    debug("Fuzzing: Returned [" + ret + ’]’);
}

                         [ ... code omitted ... ]
```

Listing 4.3: Fuzzer instrumentation code.

The rest 980 lines of code are our fuzzers implementation logic, including: mutation stages, crash detection and code execution instrumentation. All this code, must not be modified by the fuzzer’s users. In order to achieve the coverage guided feature we use Frida's **stalker** engine. Stalker is a code tracing engine that allows us to follow any thread's code execution. In our fuzzer we set up this tool in such a way that every time we are inserting in a new block of code the stalker sends us the start and end address of this block. So in each iteration we check if new blocks are revealed in contrast to previous runs and if so we save the current test case to corpus for new mutations. The clever part is not only to check for new blocks but also the order they are discovered.

4.4 Using our fuzzer

In this section, we will run our fuzzer against our vulnerable android application. As the concept for developing this fuzzer is that we are targeting third-party application that we do not have the source code we will not use it even though we have it. Firstly, we have to reverse engineer it using Jdax. Studying the output in Figure 4.5 and compared to the source code B.1 we can see how good Jdax works. At once we can identify that native code is called through JNI and the library is called nativelib.so. Also we can identify which is this native function.

```
                                            [ ... code omitted ... ]

static {
    System.loadLibrary("nativelib");
}

                                            [ ... code omitted ... ]

public native String stringFromJNI();

                                            [ ... code omitted ... ]
```

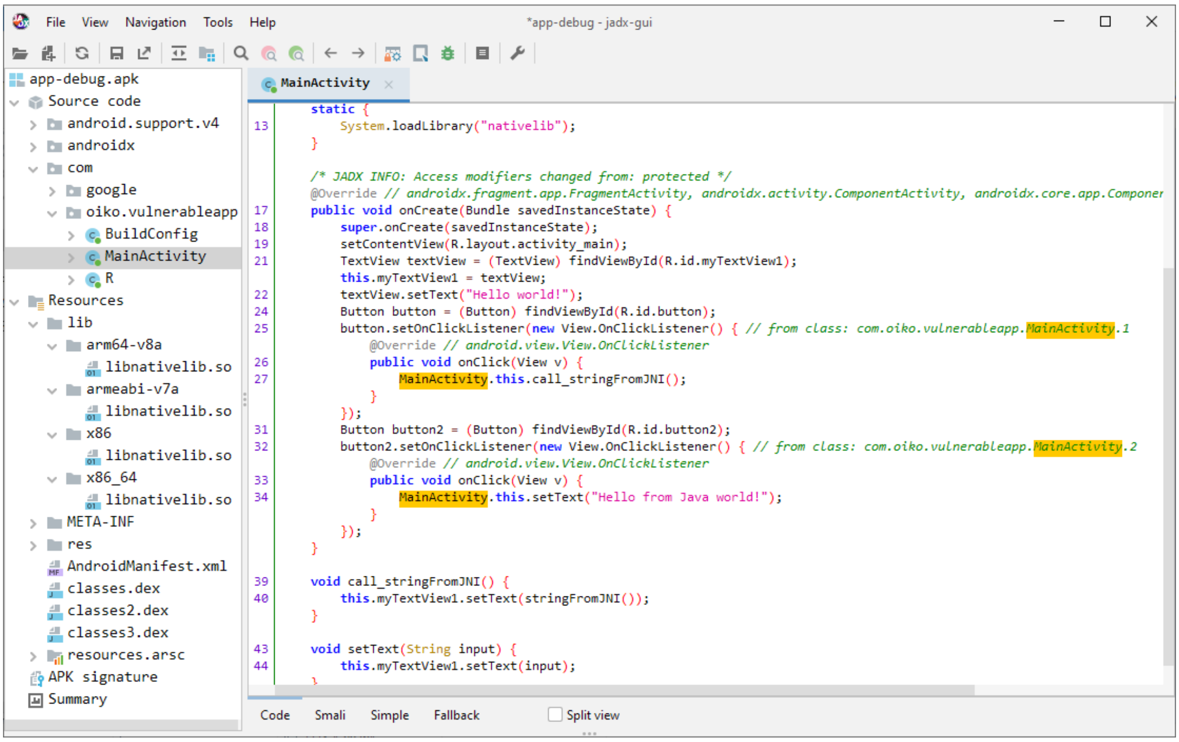
Listing 4.4: Native code use identification.

At this point thanks to the dynamic characteristics we could start fuzzing from Java part but the specific JNI function only returns data so we will investigate in more depth the native parts. In the next step we will extract this native library using apktool, Listing 4.5. Apktool will create a folder named lib containing all the extracted libraries under folders named by the architectures (arm64-v8a, armeabi-v7a, x86, x86_64) that they are targeting. Next, we will choose x86_64 library as we will run all the experiments in a x86_64 CPU architecture and using ghidra we will reverse engineer it.

```
C:\Users\****\apk>apktool.bat d app-debug.apk
I: Using Apktool 2.6.1 on app-debug.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\****\AppData\Local\apktool\framework\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Baksmaling classes3.dex...
I: Baksmaling classes2.dex...
I: Copying assets and libs...
```

```
I: Copying unknown files...
I: Copying original files...
```

Listing 4.5: Apktool output.



```java
    static {
        System.loadLibrary("nativelib");
    }

    /* JADX INFO: Access modifiers changed from: protected */
    @Override // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, androidx.core.app.Componer
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView textView = (TextView) findViewById(R.id.myTextView1);
        this.myTextView1 = textView;
        textView.setText("Hello world!");
        Button button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() { // from class: com.oiko.vulnerableapp.MainActivity.1
            @Override // android.view.View.OnClickListener
            public void onClick(View v) {
                MainActivity.this.call_stringFromJNI();
            }
        });
        Button button2 = (Button) findViewById(R.id.button2);
        button2.setOnClickListener(new View.OnClickListener() { // from class: com.oiko.vulnerableapp.MainActivity.2
            @Override // android.view.View.OnClickListener
            public void onClick(View v) {
                MainActivity.this.setText("Hello from Java world!");
            }
        });
    }

    void call_stringFromJNI() {
        this.myTextView1.setText(stringFromJNI());
    }

    void setText(String input) {
        this.myTextView1.setText(input);
```

Figure 4.5: Jdax output.

Looking Figure 4.6 we can see all the functions that Ghidra detected inside our native library. Again, the output that Ghidra gives as are amazingly with the real source code.

Next Figure, 4.7 shows the code that Ghidra produces during decompilation. In Listing 4.6 we can see the original source code.

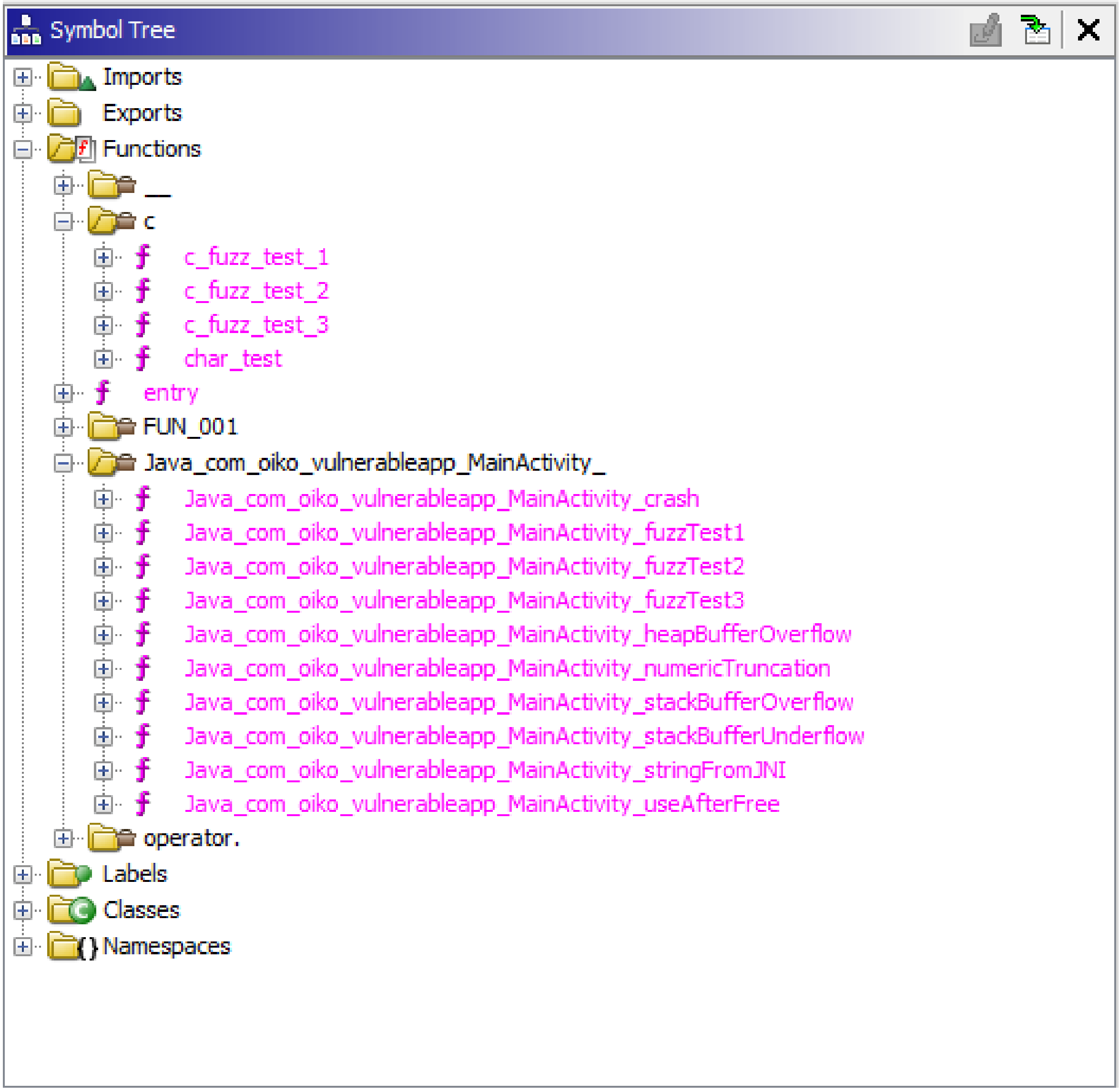
```c
extern "C"
int c_fuzz_test_3(char *buf, int len) {
    char local_buf[2];
    if (len > 0 && buf[0] == 'H')
        if (len > 1 && buf[1] == 'I' && char_test(buf[1]) == 0)
            if (len > 2 && buf[2] == '!')
                memcpy(local_buf, buf, strlen(buf));
    return 0;
}
```

Listing 4.6: Target function used only nativly.

```c
extern "C"
JNIEXPORT jint JNICALL
Java_com_oiko_vulnerableapp_MainActivity_fuzz_test_3(JNIEnv *env, jobject
     thiz, jstring param, jint len) {
    char local_buf[2];
    const char * buf= (*env).GetStringUTFChars(param, NULL);
    if (len > 0 && buf[0] == 'H')
        if (len > 1 && buf[1] == 'I')
```

```
      if (len > 2 && buf[2] == '!')
        memcpy(local_buf, buf, strlen(buf));
  return 0;
}
```

Listing 4.7: Target function called by JNI.



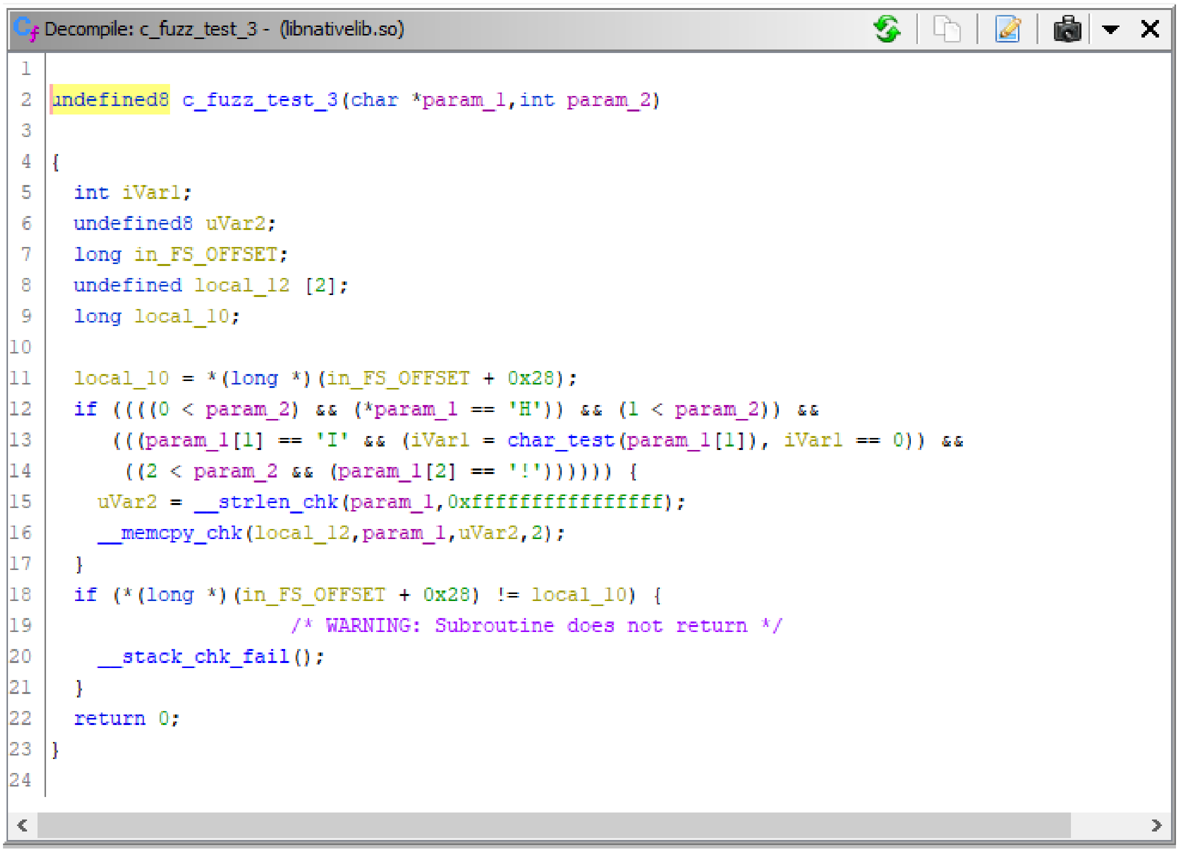
```
Symbol Tree
⊞ Imports
⊞ Exports
⊟ Functions
    ⊞ __
    ⊟ c
        ⊞ c_fuzz_test_1
        ⊞ c_fuzz_test_2
        ⊞ c_fuzz_test_3
        ⊞ char_test
    ⊞ entry
    ⊞ FUN_001
    ⊟ Java_com_oiko_vulnerableapp_MainActivity_
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_crash
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_fuzzTest1
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_fuzzTest2
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_fuzzTest3
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_heapBufferOverflow
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_numericTruncation
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_stackBufferOverflow
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_stackBufferUnderflow
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_stringFromJNI
        ⊞ Java_com_oiko_vulnerableapp_MainActivity_useAfterFree
    ⊞ operator.
⊞ Labels
⊞ Classes
⊞ Namespaces
```

Figure 4.6: Ghidra symbol tree output.

In order to fuzz out target we must have a working environment that runs an Android System. We can use either a USB connected device either android emulator. Due to the fact that we want to create a fuzzer independent from specific vendors devices we will use Android studio's included emulator.

The following commands, Listing 4.8 create the virtual environment. It starts the emulator as root user, installs our application, moves Frida server to emulator and after giving the appropriate permissions starts it.

```
C:\Users\****\> adb root
C:\Users\****\> adb install C:\Users\****\vulnerableAndroidApp\app\build\
    outputs\apk\debug\app-debug.apk
```



```
undefined8 c_fuzz_test_3(char *param_1,int param_2)

{
  int iVar1;
  undefined8 uVar2;
  long in_FS_OFFSET;
  undefined local_12 [2];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  if (((((0 < param_2) && (*param_1 == 'H')) && (1 < param_2)) &&
     (((param_1[1] == 'I' && (iVar1 = char_test(param_1[1]), iVar1 == 0)) &&
      ((2 < param_2 && (param_1[2] == '!')))))) {
    uVar2 = __strlen_chk(param_1,0xffffffffffffffff);
    __memcpy_chk(local_12,param_1,uVar2,2);
  }
  if (*(long *)(in_FS_OFFSET + 0x28) != local_10) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

Figure 4.7: Ghidra function decompilation.

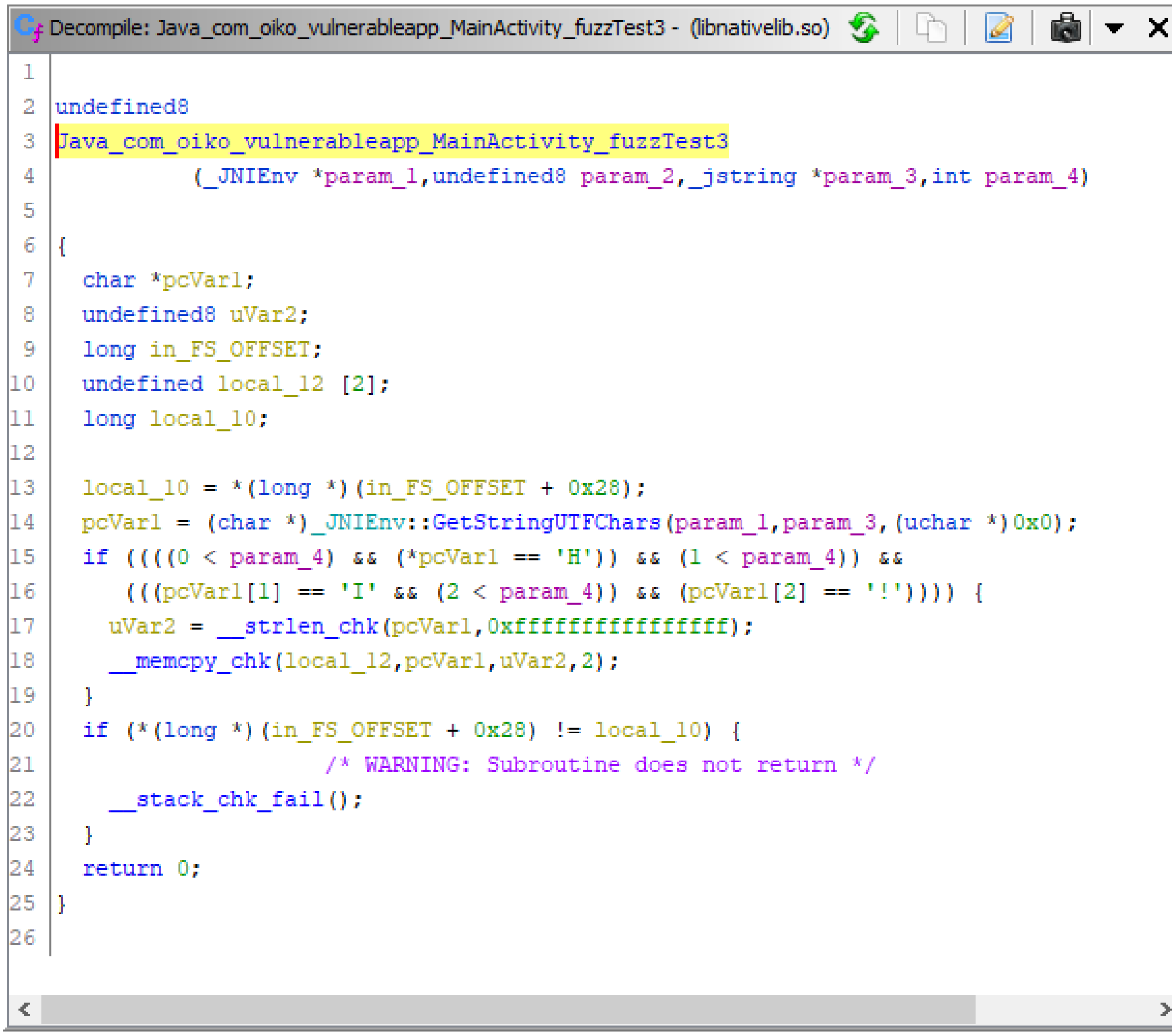
```
C:\Users\****\> adb push "%~dp0\frida-server" /data/local/tmp/
C:\Users\****\> adb shell "chmod 755 /data/local/tmp/frida-server"
C:\Users\****\> adb shell "/data/local/tmp/frida-server &"
```

Listing 4.8: Android emulator setup.

Next we are setting up our javascript fuzzer for attacking c_fuzz_test_3 function as Listing 4.9 and we run Frida with our custom agent fuzzing script as shown in Listing 4.10.

```
///////////////////////////     TARGET CONFIG ///////////////////////////

var debugging_enabled           = false;
var target_module               = 'libnativelib.so';
var target_function             = 'c_fuzz_test_3';
var target_function_ret_type    = 'int';
var target_function_arg_types   = ['pointer', 'int'];
var watch_modules               = [];

var corpus                      = ['AAAAA'];

var deterministic_stage_enabled = true;
var havoc_stage_enabled         = true;

                        [ ... code omitted ... ]
```

Listing 4.9: Target Configuration part.


Decompile: Java_com_oiko_vulnerableapp_MainActivity_fuzzTest3 - (libnativelib.so)

```

undefined8
Java_com_oiko_vulnerableapp_MainActivity_fuzzTest3
          (_JNIEnv *param_1,undefined8 param_2,_jstring *param_3,int param_4)

{
  char *pcVar1;
  undefined8 uVar2;
  long in_FS_OFFSET;
  undefined local_12 [2];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  pcVar1 = (char *)_JNIEnv::GetStringUTFChars(param_1,param_3,(uchar *)0x0);
  if (((((0 < param_4) && (*pcVar1 == 'H')) && (1 < param_4)) &&
     (((pcVar1[1] == 'I' && (2 < param_4)) && (pcVar1[2] == '!')))) {
    uVar2 = __strlen_chk(pcVar1,0xffffffffffffffff);
    __memcpy_chk(local_12,pcVar1,uVar2,2);
  }
  if (*(long *)(in_FS_OFFSET + 0x28) != local_10) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}

```

Figure 4.8: Ghidra function decompilation.

```
C:\Users\****\> adb devices
List of devices attached
emulator-5554   device

C:\Users\****\> emulator -avd Pixel_5_API_33
C:\Users\****\> frida-ps -Ua
PID  Name           Identifier
4  -------------  ---------------------------------------
7596  Calendar       com.google.android.calendar
8137  Chrome         com.android.chrome
7710  Clock          com.google.android.deskclock
7816  Gmail          com.google.android.gm
1478  Google         com.google.android.googlequicksearchbox
1478  Google         com.google.android.googlequicksearchbox
1502  Messages       com.google.android.apps.messaging
7768  Phone          com.google.android.dialer
 946  SIM Toolkit    com.android.stk
8312  Settings       com.android.settings
7981  YouTube        com.google.android.youtube
5471  vulnerableApp  com.oiko.vulnerableapp

C:\Users\****\> frida -U vulnerableapp -l agent.js
```

```
     ____
    / _  |   Frida 16.0.2 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about ’object’
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Attaching...

[+] Loading script
[+] Attached module          : libnativelib.so
[+] Target function          : c_fuzz_test_3
[+] Function argument types: pointer,int
[+] Function return type    : int

[#1] Time: [0:0:0] cov: [1] exec/sec: [2] stage: [init] corp: [1/1] craches: [0]
[#91] Time: [0:0:2] cov: [1] exec/sec: [34] stage: [arith 8/8] corp: [1/1] craches: [0]
[#123] Time: [0:0:3] cov: [2] exec/sec: [35] stage: [arith 8/8] corp: [1/2] craches: [0]
[#241] Time: [0:0:6] cov: [2] exec/sec: [37] stage: [arith 8/8] corp: [1/2] craches: [0]
[#366] Time: [0:0:9] cov: [2] exec/sec: [38] stage: [arith 8/8] corp: [1/2] craches: [0]
[#448] Time: [0:0:11] cov: [2] exec/sec: [39] stage: [bitflip 1/1] corp: [2/2] craches: [0]
[#487] Time: [0:0:12] cov: [3] exec/sec: [39] stage: [bitflip 4/1] corp: [2/3] craches: [0]
[#596] Time: [0:0:15] cov: [3] exec/sec: [38] stage: [arith 8/8] corp: [2/3] craches: [0]
[#714] Time: [0:0:18] cov: [3] exec/sec: [39] stage: [arith 8/8] corp: [2/3] craches: [0]
[#831] Time: [0:0:21] cov: [3] exec/sec: [39] stage: [interest 8/8] corp: [2/3] craches: [0]

[!] Exception found! Type: [abort] Corpus: [72,73,33] [HI!]

[#912] Time: [0:0:24] cov: [3] exec/sec: [38] stage: [bitflip 2/1] corp: [3/3] craches: [1]
[#920] Time: [0:0:24] cov: [4] exec/sec: [37] stage: [bitflip 4/1] corp: [3/4] craches: [1]
```

Listing 4.10: Android emulator setup.

As we can in Figure, Frida loads our script, detects our targeted function and the fuzzing process starts. At start our corpus is the string ’AAA’, the coverage indication (cov) is 1 and corpus (corp) is 1/1. The first number indicates which corpus is used currently and the second the total length. As we are in the start, we using the first one ’AAA’ and there are only one, so it is 1/1. During mutation stage when we test the string ’HAA’ a new block of code discovered. At this point ’HAA’ will be saved in corpus so the corp indication is becoming 1/2 and cov increased by 1. When the fuzzer will finish all ’AAA’ mutations will go to use the next one which is the newly added ’HAA’. During ’HAA’ mutation when ’HIA’ will be tested a new block will be discovered and ’HIA’ will be added as new corpus. Eventually, when ’HI!’ will be tested memory violation will be caused and the function will crash our application. In this case a Exception message will be printed with the corpus that caused the specific crash.

Chapter 5

Conclusion

In this thesis, we showed how important is the fuzzing process on finding vulnerabilities in Android applications. With the growing number of Android application's usage every day, the possibility of existent and not discovered vulnerabilities is increased making it critical for users' data privacy and security. Android applications that are developed in Java are free from buffer overflow related vulnerabilities usually found into native compiled applications. But, the increased need of better performance lead to writing code running closer to the hardware by using native compiled libraries. These libraries enclosed into the applications and included as shared object files are vulnerable to memory related vulnerabilities and as native applications are prone to memory bugs.

Our approach towards finding vulnerabilities in Android application and overcome existing challenges in fuzzing in the specific area is the design of a more dynamic tool as base and easily modified by the users. At time writing this essay, there are no existing Android tools that can be used to implement black-box fuzzing and running in a kind of parallel by creating virtual machines simulating Android environments in order to fuzz Android applications or Android system. So, we developed our own tool using Frida reverse engineering framework, to achieve automated black-box fuzzing with the advantage of dynamic code execution in order to bypass authentication mechanisms or other application code that could drive our fuzzing process in failure. Our fuzzer during testing, shown that with the right instrumentation is capable of handling all of JNI calls. Expect the native part of the application as target, our tool is capable attacking also Java code.

The fuzzer designed in such a way that can target virtual Android devices running in a server or real Android devices connected the the workstation with USB connection. These type of design allow as to run our fuzzer at first against virtual devices running in a high performance computers or servers with more RAM and CPU than an actual device. Then after the vulnerability found we can pass at evaluation procedure running our fuzzer slightly modified targeting the specific vulnerability at a real device. During testing due to that fact that the majority of bugs found at deterministic stages making as changing our strategy. Our fuzzer modified in such a way the two stages of data mutation separated. As the time of finding is crucial, through a self made automation python script the fuzzer runs the deterministic mutation stage in one virtual device and on one or more virtual device the no deterministic havoc stage. The reason for that is simple. The havoc stage as it tries randomly generated mutated data if we were lucky could try at first application run a test case that the deterministic stage could try at the end after long hours of running.

Future Work

Our seed mutation approach is ineffective in structured inputs such as XML and image files. At firsts runs of testing our approach is important as it checks parts of code that make the fist evaluation of structured data but then if that parts are correct our fuzzer lose significant time trying these parts and never testing the rest code. Future work can be done in developing a grammar (semantic rules) for structure data so that fuzzers can run in parallel in new virtual devices variations of data mutation that are correct in terms of structure but not in the included information.

As we can see in Listing 4.10 during running our tool in a virtual device with 4 cores and 1.5 Gb of allocated RAM can succeed 39 executions per second. Future work can be done in rewriting some parts of our fuzzer in native code. Especially the code that has to do with data mutation and state detection. The only code that can retained unchanged will be the parts that has to do with dynamic instrumentation of targeted functions. Frida provides all the JavaScript code that can be used for injection, function manipulation and memory reading also in C language. As shown in 4.3 Frida is broken down into several modules. Frida-agent that is injected into the targeted process and controlled by Frida-core. Frida-core which contains the main injection code and is the main mechanism of control from us. It's job is to inject code into a process, creating as a result a thread running our instrumentation JavaScript code. Frida-gumjs that contains the JavaScript bindings and reading our Javascript code calls the equivalent C functions through Frida-gum. The main modules that will help as to optimize the performance are Frida-agent and Frida-gum. In our new optimized version we will do all move the mutation logic and tracing from Javascript part into frida-agent and we will create a binding to get the mutated data into Javascript part. Frida also provides a special feature that called CModule. CModule takes a string of C code, dynamically detects platform's architecture, compiles it to machine code using TinyCC and injects this straight to memory.

Appendix A

Acronyms

ABI Application Binary Interface

AFA Anti-Fragmentation Agreement

AIDL Android Interface Definition Language

APP Android Application

APK Android Package Kit

ART Android Application Runtime

ASLR Address Space Layout Randomization

CFI Control flow integrity

COM Common Object Model

CORBA Common Object Broker Request Architectures

CTS Compatibility Test Suite

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

DTBO Device Tree Blob Overlay

FBE File-based encryption

GKI Generic Kernel Image

GNSS Global Navigation Satellite System

HAL Hardware Abstraction Layer

IMS IP Multimedia Subsystem

IPS Inter Process Communication

JDK Java Development Kit

JVM Java Virtual Machine

LAN Local Area Network

NDK Native Development Kit

NIST National Institute of Technology

NVD National Vulnerability Database

OS Operating System

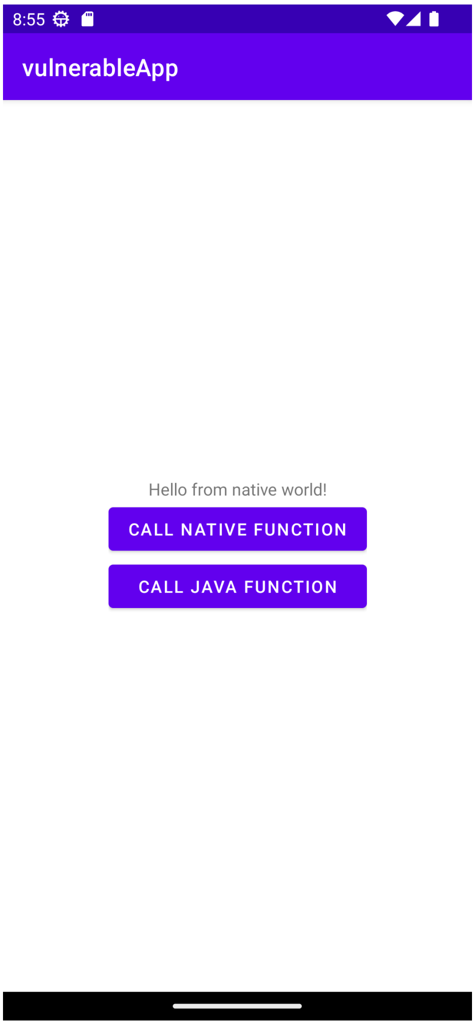
RPC Remote Procedure Call

SDK Software Development Kit

VNDK Vendor Native Development Kit

Appendix B

Vulnerable android application



8:55

vulnerableApp

Hello from native world!

Figure B.1: Application's UI.

```
package com.oiko.vulnerableapp;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    TextView myTextView1;

    static {
        System.loadLibrary("nativelib");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        myTextView1 = findViewById(R.id.myTextView1);
        myTextView1.setText("Hello world!");

        Button button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                call_stringFromJNI();
            }
        });

        Button button2 = (Button) findViewById(R.id.button2);
        button2.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                setText("Hello from Java world!");
            }
        });
    }

    void call_stringFromJNI(){
        myTextView1.setText(stringFromJNI());
    }

    void setText(String input){
        myTextView1.setText(input);
    }

    public native String stringFromJNI();
}
```

Listing B.1: Java application code.

```
#include "nativelib.h"

extern "C"
JNIEXPORT jstring JNICALL
Java_com_oiko_vulnerableapp_MainActivity_stringFromJNI(JNIEnv *env,
    jobject thiz) {
    std::string hello = "Hello from native world!";
    return env->NewStringUTF(hello.c_str());
}

extern "C"
JNIEXPORT jstring JNICALL
Java_com_oiko_vulnerableapp_MainActivity_crash(JNIEnv *env, jobject thiz)
     {
```

```
    char buf[1];
    char * src = "crash";
    strncpy(buf, src, sizeof(src));
}

extern "C"
JNIEXPORT void JNICALL
Java_com_oiko_vulnerableapp_MainActivity_stack_buffer_underflow(JNIEnv *env, jobject thiz, jstring param) {
    char buf[12];
    const char * input= (*env).GetStringUTFChars(param, NULL);
    strncpy(buf, input, sizeof(buf));
    size_t length = strlen(buf);
    int index = (length - 1);
    while (buf[index] != ':') {
        buf[index] = '\0';
        index--;
    }
}

extern "C"
JNIEXPORT void JNICALL
Java_com_oiko_vulnerableapp_MainActivity_stack_buffer_overflow(JNIEnv *env, jobject thiz, jstring param) {
    char buf[5];
    const char * input= (*env).GetStringUTFChars(param, NULL);
    strcpy(buf, input);
}

extern "C"
JNIEXPORT void JNICALL
Java_com_oiko_vulnerableapp_MainActivity_heap_buffer_overflow(JNIEnv *env, jobject thiz, jstring param) {
    int BUFSIZE = 10;
    char * buf = static_cast<char *>(malloc(sizeof(char) * BUFSIZE));
    const char * input= (*env).GetStringUTFChars(param, NULL);
    strcpy(buf, input);
    free(buf);
}
extern "C"
JNIEXPORT void JNICALL
Java_com_oiko_vulnerableapp_MainActivity_use_after_free(JNIEnv *env, jobject thiz, jstring param) {
    int BUFSIZE = 10;
    char * buf = static_cast<char *>(malloc(sizeof(char) * BUFSIZE));
    const char * input= (*env).GetStringUTFChars(param, NULL);
    strncpy(buf, input, strlen(input));
    free(buf);
    strncpy(buf, input, strlen(input));
}

extern "C"
JNIEXPORT void JNICALL
Java_com_oiko_vulnerableapp_MainActivity_numeric_truncation(JNIEnv *env, jobject thiz) {
    unsigned int buf1_len = UINT_MAX - 256;
    unsigned int buf2_len = 256;
    unsigned int buf3_len = 256;

    char buf1[buf1_len], buf2[buf2_len], buf3[buf3_len];
    int buf3_new_len = (buf1_len + buf2_len);

    if (buf3_new_len <= 256) {
        memcpy(buf3, buf1, buf1_len);
        memcpy(buf3 + buf1_len, buf2, buf2_len);
    }
}

extern "C"
JNIEXPORT jint JNICALL
Java_com_oiko_vulnerableapp_MainActivity_fuzz_test_1(JNIEnv *env, jobject thiz, jstring param, jint len) {
    char local_buf[15];
    const char * buf= (*env).GetStringUTFChars(param, NULL);
    if (len > 0 && buf[0] == ’H’)
        if (len > 1 && buf[1] == ’I’)
            if (len > 2 && buf[2] == ’!’)
                return 1;
    return 0;
}

int char_test(char a){
    if (a == ’a’) {
        return 1;
    } else {
        if (a == ’I’) {
            return 0;
        }
        return 8;
    }
}

extern "C"
JNIEXPORT jint JNICALL
Java_com_oiko_vulnerableapp_MainActivity_fuzz_test_2(JNIEnv *env, jobject thiz, jstring param, jint len) {
    char local_buf[15];
    const char * buf= (*env).GetStringUTFChars(param, NULL);
    if (strlen(buf) == 4564) return 2;

    if (strlen(buf) == 3 && buf[5] == ’B’) return 1;

    if (buf[5] == ’C’) return 1;

    if (strlen(buf) > 0 && buf[0] == ’X’ && buf[1] == ’d’) {
        memcpy(local_buf+16, buf, strlen(buf));
    }

    if (char_test(’u’) > 1) {
        return 4;
    };

    return 1;
}
extern "C"
JNIEXPORT jint JNICALL
```

```
Java_com_oiko_vulnerableapp_MainActivity_fuzz_test_3(JNIEnv *env, jobject thiz, jstring param, jint len) {
    char local_buf[2];
    const char * buf= (*env).GetStringUTFChars(param, NULL);
    if (len > 0 && buf[0] == 'H')
        if (len > 1 && buf[1] == 'I')
            if (len > 2 && buf[2] == '!')
                memcpy(local_buf, buf, strlen(buf));
    return 0;
}

extern "C"
int c_fuzz_test_1(char *buf, int len) {
    char local_buf[2];
    if (len > 0 && buf[0] == 'H')
        if (len > 1 && buf[1] == 'I')
            if (len > 2 && buf[2] == '!')
                memcpy(local_buf, buf, strlen(buf));
    return 0;
}

extern "C"
void c_fuzz_test_2(char *buf, int len) {
    char local_buf[13];
    if (strlen(buf) == 4564) return;

    if (strlen(buf) == 3 && buf[5] == 'B') return;

    if (buf[5] == 'C') return;

    if (strlen(buf) > 0 && buf[0] == 'X' && buf[1] == 'd') {
        memcpy(local_buf+16, buf, strlen(buf));
    }

    if (char_test('u') > 1) {
        return;
    };
}

extern "C"
int c_fuzz_test_3(char *buf, int len) {
    char local_buf[2];
    if (len > 0 && buf[0] == 'H')
        if (len > 1 && buf[1] == 'I' && char_test(buf[1]) == 0)
            if (len > 2 && buf[2] == '!')
                memcpy(local_buf, buf, strlen(buf));
    return 0;
}
```

Listing B.2: Native's library code.

Bibliography

[1] Zerodium. The premium exploit acquisition platform. [Online]. Available: https://zerodium.com/

[2] Google. Android os architecture. [Online]. Available: http://https://developer.android.com/guide/platform

[3] B. Nongpoh, "Registered report: Fine-grained coverage-based fuzzing," *in International Fuzzing Workshop (FUZZING)*, 2022.

[4] Skylot. jadx - dex to java decompiler. [Online]. Available: https://github.com/skylot/jadx

[5] Frida. Frida - dynamic instrumentation toolkit. [Online]. Available: https://github.com/frida/frida

[6] Statcounter. Statcounter global stats. [Online]. Available: https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide/#monthly-202009-202209

[7] Zimperium. 2022 global mobile threat report. [Online]. Available: https://www.zimperium.com/global-mobile-threat-report/

[8] Google. Google android: Cve security vulnerabilities, versions and detailed reports. [Online]. Available: https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224

[9] NIST. Vulnerability metrics. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss

[10] Google. Zero-day exploits in the wild. [Online]. Available: https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/edit#gid=1662223764

[11] ——. Google security documentation. [Online]. Available: https://source.android.com/docs/security

[12] Apache. Apache harmony project. [Online]. Available: https://harmony.apache.org/

[13] P. Inc. Openbinder project. [Online]. Available: http://www.angryredplanet.com/~hackbod/openbinder/docs/html/

[14] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing," *in Proc. Int. Workshop Random Test., pp. 46-54*, 2006.

[15] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *An empirical study of the reliability of UNIX utilities*, 1990.

[16] B. P. Miller and al., "Fuzz revisited: A re-examination of the reliability of unix utilities and services," *in Dept. Comput. Sci., Univ. Wisconsin-Madison, Madison, WI, USA, Tech. Rep. 1268*, 1995.

[17] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," *in Proc. 4th Conf. USENIX Windows Syst. Symp., Seattle, WA, USA, vol. 4, pp. 1-10*, 2000.

[18] J. Offutt and J. Hayes, "A semantic model of program faults," *in Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis), pages 195-200, San Diego*, 1996.

[19] Z. N. J. Peterson and C. Miller, "Analysis of mutation and generation-based fuzzing," *in Tech. rep. ISE Independent Security Evaluators*, 2007.

[20] Y. Li and al, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," *in arxiv Computer and Science, Cryptography and Security*, 2020.

[21] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," *in 14th International Symposium on Software Reliability Engineering pp. 99-107*, 2003.

[22] P. Ammann and J. Offutt, "Introduction to software testing," *in Cambridge University Press*, 2016.

[23] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. D. Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," *in 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1-10*, 2010.

[24] Anestisb. Android port of radamsa. [Online]. Available: https://github.com/anestisb/radamsa-android

[25] Google. Honggfuzz. [Online]. Available: https://github.com/google/honggfuzz

[26] Anestisb. melkor-android. [Online]. Available: https://github.com/anestisb/melkor-android

[27] Fuzzing. Mffa - media fuzzing framework for android. [Online]. Available: https://github.com/fuzzing/MFFA

[28] AFLplusplus. American fuzzy lop plus plus (afl++). [Online]. Available: https://github.com/AFLplusplus/AFLplusplus

[29] Google. Libfuzzer. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[30] IBotPeaches. Android apps reverse engineering. [Online]. Available: https://github.com/iBotPeaches/Apktool

[31] Ghidra. Software reverse engineering framework. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra

[32] AFL. American fuzzy lop. [Online]. Available: http://lcamtuf.coredump.cx/afl/