University of Piraeus

School of Information and Communication Technologies

Department of Digital Systems

Postgraduate Program of

Studies MSc Digital Systems

Security

Title: Android malware network traffic detection using visual representation (AF)

Supervisor Professor: Christos Xenakis

Name-Surname	E-mail	Student ID.
MATTHAIOS STAVROU	mstauroy@gmail.com	mte2030

Piraeus

01/11/2022

This essay is dedicated to my mother who passed away recently

Abstract

There is a growing concern among mobile device users worldwide about Android malware. An effective method of detecting and analyzing Android malware is through the analysis of the network traffic that is generated because of the malware's operation. A malware analysis can provide valuable insights into the inner workings of the malware, as well as its intended targets and potential impacts to the user. Our goal in this paper is to present a novel approach for detecting Android malware network traffic using visual representation techniques to detect Android malware network traffic. As part of our approach, we utilize advanced data visualization techniques to display network traffic data in a clear, effective, and efficient manner, allowing for the efficient and accurate analysis of Android malware network activity.

In our presentation, we will present the developer tool, which is a script that allows users to automate Frida commands and easily retrieve information regarding an app's classes, properties, and methods. It will also provide users with a script tool which utilizes the Android Debug Bridge (adb) command-line tool to perform a variety of actions related to Android applications.

It is our belief that our approach will be a valuable tool for those researchers and analysts who are working to prevent Android malware from spreading throughout the world.

Table of Contents

1. Android Security Process

a. Introduction

Android is a widely used mobile operating system that has a comprehensive security model in place to protect users and their devices from malicious software and security threats. The Android security model incorporates various layers of protection that work together to secure the device and keep it safe. The core of the Android security model is the Android operating system itself, which is designed with security in mind and includes features such as process isolation, secure IPC, and application sandboxing to protect against vulnerabilities and malware.

To ensure the security of devices and apps, Google has implemented a set of security policies and practices for Android device manufacturers and developers. These policies and practices ensure that devices and apps are secure and meet the highest standards for security and privacy. One of the key ways that Android maintains security is through regular security updates, which address vulnerabilities and fix bugs that could be exploited by malicious actors. Google works with device manufacturers and carriers to ensure that these updates are rolled out to users in a timely manner.

In addition to the security measures implemented by Google and device manufacturers, users also have a number of tools at their disposal to help secure their devices. These include the ability to set up screen locks and use biometric authentication, as well as the ability to download and use security apps from the Google Play Store. Overall, the Android security model is designed to keep users and their devices safe from threats and vulnerabilities. By following best practices and keeping devices up to date with the latest security updates, users can help ensure that their devices remain secure.

As with other app-stores, Google Play requires that applications be digitally signed by their developers. Signing an application is intended to identify the author of the application and establish trust between the developer and Google. The secret key used for signing is known only to the developer, which means that no one else can sign the application with that key, ensuring that the developer knows that the presence of his signature indicates that the application has not been compromised. It also ensures that the developer can be held accountable for the application's behavior.

The Android Application Sandbox is defined by the app signature for each application. The certificate identifies which user ID is associated with which application. As a result, different applications run with different user IDs. Any application installed on the device is verified for proper signatures by the package manager. It is possible for the application that is being installed to share data if the public key in the certificate matches any key used to sign other applications on the device a user ID that is shared with other apps signed with the same public key. It must be specified in the manifest file of the application being installed. The certificates in Android applications are not verified by CAs. Therefore, applications may be signed by a third party or even by themselves. Furthermore, applications are able to declare security permissions based on the signature they bear. When two applications are signed with the same key, they are developed by the same author, and thus a shared Application Sandbox can be enabled using the shared user ID feature.

In conclusion, the Android security model is designed to keep users and their devices safe from threats and vulnerabilities. Through the use of various security features and regular security updates, as well as tools and policies implemented by Google and device manufacturers, users can help ensure the security of their devices. Additionally, the use of digital signatures and the Android Application Sandbox helps to establish trust and accountability between developers and users, and provides an added layer of security for the device [1].

b. Application Sandbox

The Android operating system incorporates a security feature known as an Application Sandbox. This feature allows for the isolation and protection of individual apps from each other and from the system. As a virtualization environment, it gives each app the ability to run as if it were the only app on the device with its own private resources and virtual file system. This means that apps will be prevented from accessing or modifying sensitive data belonging to other apps or the system itself, if this feature is enabled.
The Android application sandbox is an essential aspect of the Android security model, designed to protect users and their devices from malicious software and other threats. The Android platform makes use of Linux user-based protection as a means of identifying and isolating app resources. This results in the separation of apps and also serves to protect the system from malicious applications by isolating the apps from one another.

To achieve this, each Android application is assigned a unique user ID (UID). The Android application is then run in its own process, which is accomplished by using standard Linux facilities such as user and group IDs. The kernel enforces security between apps and the system at the level of the processes, by enforcing security between apps and the system. As a default, apps cannot interact with each other or access the operating system by themselves, and are not able to interact with each other by default.
This security model extends to both native code and OS applications, due to the Application Sandbox's location in the kernel. Application Sandboxes contain all software above the kernel, such as OS libraries, application frameworks, application runtimes, and all applications. Depending on the platform, developers may be restricted to a specific development framework, API set, or programming language. However, a native application is as sandboxed as an interpreted application on Android, as there are no restrictions on how it can be written to ensure security.
In conclusion, the Android Application Sandbox is an essential security feature that allows for the isolation and protection of individual apps from each other and from the system. It makes use of Linux user-based protection to identify and isolate app resources and assigns each app a unique user ID for this purpose. It also enforces security between apps and the system at the level of the processes, making it a highly secure and efficient feature for Android devices [2].

I. Protections

The security of the Linux kernel in Android devices is protected by the use of an Application Sandbox. While individual protections may not be foolproof, implementing defense-in-depth strategies is essential to prevent vulnerabilities from compromising the operating system or other applications. The Android Application Sandbox enforces several protections to strengthen the security of the system.
In Android 5.0, SELinux was introduced to provide mandatory access control (MAC) separation between the system and apps. The UID-based discretionary access control (DAC) sandbox was also used to enforce inter-app isolation. In Android 6.0, the SELinux sandbox was extended to isolate apps across the boundaries of each physical user. Additionally, safer defaults for application data were set, with permissions on an app's home directory changed from 751 to 700 for apps with targetSdkVersion >= 24.

In Android 8.0, a seccomp-bpf filter was introduced to limit the syscalls apps were allowed to use, further strengthening the app/kernel boundary. In Android 9, all non-privileged apps with a targetSdkVersion greater than 28 were required to run in SELinux sandboxes, with MACs provided per application. This not only improves app separation, but also prevents apps from overriding safe defaults and making their data accessible to others.

Finally, in Android 10, direct access to /sdcard/DCIM was removed, providing a limited raw view of the filesystem. This allows apps to access their package-specific paths directly via any method, such as Context.getExternalFilesDir(). Overall, the Android Application Sandbox enforces a variety of protections to enhance the security of the Linux kernel and protect user data [2].

II. Guidelines for sharing files

The When creating mobile apps, it's essential to consider the security risks that come with sharing files. One of the most prevalent poor practices is making app data publicly accessible, as it grants access to anyone and doesn't allow limiting access to specific recipients. This can lead to information leaks, vulnerabilities and makes the app a target for malware.

Instead of this approach, it's recommended to follow these guidelines:

- Use a content provider when your app needs to share files with another app. Content providers allow sharing data with the right level of detail and don't have the same limitations as UNIX permissions that are accessible to everyone.
- For files that need to be publicly accessible such as images, videos, and audio files, store them using the MediaStore class. This class uses runtime permissions to control access to strongly-typed collections and allows using intents like ACTION_OPEN_DOCUMENT to access weakly typed files like PDFs.
- For apps targeting Android 9 and below, the requestLegacyExternalStorage manifest attribute is set to true by default. For apps targeting Android 10, the default value is false, to temporarily disable the filtered storage view, set the attribute to true.
- To enhance security, use restricted permissions and installer whitelisting. This allows only specific apps to be stored outside the sandbox, providing an extra layer of protection for apps that aren't whitelisted.

In conclusion, when sharing files in mobile apps, it's crucial to avoid making app data publicly accessible, as it can lead to security vulnerabilities. Instead, developers should use content providers, MediaStore class, and restricted permissions to share files with the right level of detail and control access. Additionally, apps targeting Android 10 should use the requestLegacyExternalStorage manifest attribute and follow best practices for app permissions.

c. SELinux

I. Introduction

In SELinux, what is explicitly allowed is listed, while everything else is forbidden. There are various modes, Enforcing, where permissions are enforced and logged, Permissive, where permissions are only logged, and Disabled, where it is disabled. SELinux is enabled on all Android devices (since Android 5.0 [3]) and set to enforce. Consequently, any finding of SELinux in any other state indicates that the environment has been modified. As part of our implementation, we employ three different methods for determining the state of SELinux. We once again rely on Android properties to provide us with information through ro.build.selinux and the RO.boot.selinux file. The first one reports the status from the boot.img about SELinux on the current build, while the second one is to see if the Kernel uses the permissive parameter. The third method involves identifying the contents of the "enforce" file in order to detect SELinux. SELinux's current state is contained in this file. Although this method is known for Linux systems, we were unable to find any relevant detection mechanism implemented in Android. The first step is to read /proc/filesystems, which contains the kernel-compiled filesystems. Those whose kernel modules are currently loaded. As a result, when reading the list of compiled filesystems. The kernel can be examined to determine if the selinuxfs is present, which would indicate that SELinux has been loaded. This method has a twofold importance, first, it matters whether we can read at all the /proc/filesystems and secondly if we find the selinuxfs. According to our reseach and testing after SELinux blocks reading several /proc/ locations after Android 8, including the filesystems 1 location. In the case we are able to read /proc/filesystems we know that SELinux is probably disabled or permissive. Alternatively, for older versions of Android, after we read the contents of /proc/filesystems, we try to detect the value selinuxfs, and if we find it we know SELinux is running on the device. Though, we do not know in which state SELinux is at this point.

In order to determine that we read the /proc/mounts and track selinuxfs, so we can extract the absolute path, which we use to find a file of interest called enforce. As a last step we read this file and in the case the content is "1" the SELinux is in enforcing state and if it is "0" the state is permissive. This method proves to be stable and more reliable than the first two according to the results.

II. Security-Enhanced Linux in Android

With regard to Security-Enhanced Linux (SELinux), Android uses mandatory access control (MAC) for all processes, including those running with root or superuser privileges (Linux capabilities). A number of companies and organizations have contributed to Android's implementation of SELinux. The SELinux operating system allows Android to secure and restrict system services, control access to application data and secure and restrict system services, and control access to application data and secure and restrict system services. Control access to application data and secure and restrict system services, control access to application data and secure and restrict system services, reduce malicious software effects, protect users from potential vulnerabilities, control application data and logs, and reduce malicious software effects. In SELinux, anything that is not explicitly allowed is denied by default. The SELinux system can operate in two modes at the global level:

- *A permissive mode is one in which permission denials are logged but not enforced.*
- *Permission denials are logged and enforced in the enforcing mode.*

In addition to SELinux in enforcing mode, Android includes a corresponding security policy that is enabled by default across the entire AOSP project. Enforcing mode prevents disallowed actions and logs all attempted violations to dmesg and logcat. Before enforcing SELinux policies, refine your software and SELinux policies with these errors [3].

SELinux also supports a per-domain permissive mode, allowing specific domains (processes) to be made permissive while leaving the rest of the system in global enforcing mode. The domain is simply a label that identifies a process or set of processes within a policy, where a process that is labeled with the same domain is treated the same. The per-domain permissive mode facilitates the incremental implementation of SELinux in an ever-increasing portion of the system, as well as the development of new policies for new services (while keeping the rest of the system enforced) [3].

III. Background

The Android security model is partly based on the concept of application sandboxes. There are a number of different sandboxes within Android. Each application runs in its own sandbox. It was necessary to assign each application a unique Linux UID when it was installed before Android 4.3 to define these sandboxes. As of Android 4.3 and later versions, SELinux has been used to further define the sandbox boundaries.

- With Android 5.x and later, everything is in enforcing mode.
- There should be no other processes running in the init domain other than init.
- A generic denial (for a block_device, socket_device, or default_service) indicates that the device requires a special domain.

In Android 6.0, the permissiveness of our policy was reduced to include better isolation between users, IOCTL filtering, reduced threat of exposed services, and further tightening of SELinux domains.

SELinux configuration has been updated in Android 7.0 to further secure the application sandbox and reduce the attack surface. As part of this release, the monolithic mediaserver stack has been split up into smaller processes in order to reduce the scope of their permissions. Check out Protecting Android with more Linux kernel defenses and Hardening the media stack for more information.

It has been updated to include SELinux as part of the Android system framework in Android 8.0, which separates lower-level vendor code from the Android system framework with the Treble feature, which separates the lower-level vendor code from the Android system framework. SELinux policy has been updated in this release to allow manufacturers and SOC vendors to update their parts of the policy, to make their images (vendor.img, boot.img, etc.), and to update those images independently of the platform they are running on.

While it is possible to run higher/newer platform (framework) versions on the device, the opposite case is not supported. Vendor images (vendor.img/odm.img) cannot be updated to a newer version than the platform (system.img). A newer platform version may introduce SELinux compatibility issues since the platform SELinux policy is at a newer version than the vendor SELinux procedure. To prevent unnecessary simultaneous updates, Android 8.0 provides a method of retaining compatibility.

d. Android Permissions

Prior to Marshmallow (API 23), permissions were much simpler to manage. During the installation process, all permissions were handled automatically during the process of installing the app. The user was presented with a list of permissions which were required by the app when installing it from the Google Play Store (some have referred to this as the "wall of permissions"). In order to install the app, the user needed to accept all of the permissions, or he or she could decide not to install it. In the past, there was no way to grant an app only certain permissions and no way for the user to revoke any permissions after it had been installed. There was no way to grant the app only certain permissions or revoke certain permissions [10]. Android permissions are used to protect user privacy and control the capabilities of apps installed on an Android device. Upon installing an app from Google Play, you may be prompted to grant the app access to certain features or data on your device. For example, a camera app might request access to the device's camera, or a weather app might request access to your location data. You can choose which permissions to grant or deny when you install the app, and you can also change your app permissions from the device's settings menu at any time. It's important to be cautious about granting app permissions, as some apps may request access to features or data that are not directly related to their intended functionality.

I. Potentially Harmful Applications (PHAs)

Potentially Harmful Applications (PHAs) are applications that may harm users, their data, or their devices. Malware is often used to refer to these applications. In addition to trojans, phishing, and spyware apps, Android has developed a wide range of categories for different types of PHAs. Categories are continuously updated and added [2].

Is it potentially harmful?

As a result of the ambiguity of the word potentially when used to describe malicious applications, there is some confusion. Whenever an app has been flagged as Potentially Harmful, Google Play Protect removes it because it contains malicious behavior, and not because we are unsure whether it is harmful or not. As malicious apps may operate differently depending on a variety of variables, an app that is harmful for one Android device may not pose a risk for another. Therefore, the term potentially is used here because

malicious apps operate differently depending on a variety of factors. It is not possible to install malicious apps to a device running the latest version of Android, however a device running an early version of Android may be vulnerable to malicious apps that use deprecated APIs to perform malicious actions. Devices that are connected to mobile service providers are at risk of mobile billing fraud, but devices that are connected only to WiFi are not affected by this type of fraud.
An application is flagged as a PHA if it clearly poses a risk to some or all Android devices and users.

PHAs requested by users

Some apps that can weaken or disable Android security features are not considered PHAs. Users are able to access functionality that they desire through these apps, such as rooting their devices and other development features. Although these apps may be harmful, users install them intentionally, so Google Play Protect manages them differently than other PHAs.
As soon as a user begins installing an app that is classified as user-wanted, Google Play Protect warns them of the potential hazards associated with that app. Due to this, the decision to install or not depends on the individual. The user-wanted classifications prevent Google Play Protect from sending additional warnings after installation, so there is no interruption in the user's experience.

The classifications

A number of categories are available for classifying PHAs so that Play Protect is able to detect them and determine the appropriate course of action. Malicious apps such as trojans, spyware, and phishing apps are included in this category, as well as user-wanted apps. Play Protect displays a warning if it detects a PHA. When Play Protect detects malicious apps, it automatically disables or removes them. When Play Protect detects that a PHA contains features from multiple categories, it classifies the app according to its most harmful characteristics. Verify Apps, for instance, identifies an app as ransomware if it applies to both spyware and ransomware categories.

II. Malware categories

A malicious piece of software is installed by a threat actor in order to cause mayhem on an organization or individual. When you (or your employees) visit a variety of different sites on the internet, malware can sometimes be found attached to emails, embedded in fraudulent links, hidden in ads, or lying-in wait on various websites that you (or your employees) might visit. Malware's primary objective is to harm or exploit computers and networks to steal data or money [11].

Keep in mind that Android's malware policy is straightforward: the Android ecosystem, including the Google Play Store, and the user devices should be free of malicious behavior (for example, malware). Our company adheres to this fundamental principle to ensure that our users and their Android devices are protected from any harmful attacks.

The term "malware" refers to any code that poses a risk to a user, their data, or their device. Continually update and add new categories of malware, such as potentially harmful applications (PHAs), binaries, and framework modifications, such as trojans, phishing, and spyware.

Malware may vary in type and capabilities, but it typically serves one of the following purposes:

- The integrity of the user's device is compromised.
- Control the device of a user.
- Provide an attacker with the ability to access, use, or otherwise exploit an infected device via remote control.
- Without adequate disclosure and consent, transmit personal data or credentials off the device.
- Infect other devices or networks through the transmission of spam or commands from the infected device.
- Fraudulently defraud the user.

An app, binary, or framework modification can generate malicious behavior even if it was not intended to do so. The reason for this is that apps, binaries, or framework modifications can perform differently based on a variety of factors. As a result, what is harmful to one Android device may not pose any risk to another Android device. The latest version of Android, for example, is not affected by malicious apps that use deprecated APIs to perform malicious behavior, however, a very early version of Android may be vulnerable. If applications, binaries, or framework modifications clearly pose a risk to Android devices and users, they are flagged as malware or PHA.

Here are the categories of malware we believe represent our core belief that users, companies, and governments should understand how their devices are being exploited and promote a secure ecosystem that supports robust innovation and provides trusted user experiences. experiences.

Backdoor

Backdoor in android refers to a method that allows unauthorized access to an android device by bypassing the normal authentication method, this could be done through various methods like installing a malicious app that acts as a backdoor, exploiting vulnerabilities in the operating system, or using advanced hacking techniques such as phishing or social engineering to trick the user into giving away their login credentials.
Backdoors on Android devices can be particularly dangerous because of the sheer number of devices that run the Android operating system. According to a recent report from Google, there are over 2.5 billion active Android devices in use worldwide, making it a prime target for attackers looking to gain unauthorized access to sensitive information.
One example of a recent Android backdoor was "Joker" discovered in early 2020, Joker malware was distributed through third-party app stores and disguising itself as legitimate apps. Once the malware is installed, it creates a backdoor on the device that allows the attackers to remotely access and control the device, as well as steal sensitive information.
Keeping an eye out for backdoors on Android devices and to take steps to protect yourself. This can include keeping your device and apps up to date, being cautious of unfamiliar apps and links, and using a mobile security app to scan for and remove any malicious software [12].
A backdoor describes a way in which a device can be accessed for potentially harmful purposes and is therefore not entirely aligned with categories like billing fraud or commercial spyware. Due to this, Google Play Protect treats backdoors as vulnerabilities when certain conditions apply.

Billing fraud

Billing fraud in the context of Android refers to the unauthorized charges made to a user's account using Android apps or the Google Play Store. This type of fraud can occur in a

variety of ways, but often involves the use of malware or malicious apps that are designed to trick users into granting access to their payment information.
One common method of billing fraud on Android involves the use of fake apps that are designed to look like legitimate ones. These apps may be designed to mimic popular apps, such as games or social media platforms, and may be offered for download through third-party app stores or websites. Once downloaded, these apps may request access to a user's payment information and may then use this information to make unauthorized charges.
Another method of billing fraud on Android involves the use of malware that is designed to infect a user's device and gain access to their payment information. This type of malware may be delivered through a variety of means, such as phishing emails or text messages, and may be disguised as a legitimate app or update. Once installed, the malware may be able to access a user's payment information and make unauthorized charges.
To protect yourself from billing fraud on Android, it is important to only download apps from the Google Play Store or other reputable sources, and to be cautious of any apps that ask for access to your payment information. It is also important to keep your device and apps up to date, as this can help to protect against known vulnerabilities that may be exploited by fraudsters. Additionally, you can use mobile security software to detect and remove malware from your device [13]. The three types of mobile billing fraud are SMS fraud, Call fraud, and Toll fraud.

Fraudulent SMS messages

Android users are often targeted by fraudulent SMS messages, also known as smishing. The messages are meant to trick users into giving them sensitive information, such as login credentials or financial information, or to download malware.
Phishing is among the most common tactics used in fraudulent SMS messages. The message may appear to come from a legitimate source, such as a bank or financial institution, asking for personal information or clicking a link. The link may lead to a fake website designed to steal the user's personal information.
Another tactic used in fraudulent SMS messages is the use of malware. These messages may contain a link that, when clicked, downloads malware onto the user's device. The malware may be used to gain access to sensitive information, such as login credentials or financial information, or to control the device remotely.

It is important to be cautious of any SMS message that asks for personal information or contains links to avoid fraudulent messages. The links in a message should not be clicked on unless you are sure they are legitimate. You should also be aware of the phone numbers or short codes from which you are receiving messages. Be cautious of unfamiliar phone numbers or short codes, and don't respond to them.

Additionally, you can install mobile security software on your device, which can help to detect and block fraudulent SMS messages, as well as other types of mobile threats. Also, it's important to keep your device and apps up to date, as this can help to protect against known vulnerabilities that may be exploited by fraudsters [14].

Fraudulent telephone calls

As a form of mobile fraud, fraudulent telephone calls, also known as vishing, target Android users. They attempt to trick them into providing sensitive information, such as login credentials or financial details. In order to scam the person, the callers may pose as representatives of legitimate organizations, such as banks or government agencies.

The caller falsifies the caller ID of the caller to make it appear as if it was coming from a legitimate organization, one of the tactics used in fraudulent telephone calls. The caller may also use high-pressure tactics to make the user feel that the call is legitimate or not. This can make it difficult for the user to determine whether the call is legitimate.

Another tactic used is the use of pre-recorded messages, also known as robocalls, these messages are automated and often use a recorded voice to deliver the message, in some cases, these calls may come with a live operator that is waiting to answer questions or collect personal information.

To protect yourself from fraudulent telephone calls, it's important to be cautious of any unsolicited calls, especially if they ask for personal or financial information. Do not provide any personal information over the phone unless you are certain that the call is from a legitimate organization, and you initiated the call. If you receive a suspicious call, you should hang up and contact the organization directly using a trusted phone number, such as one listed on the organization's official website.

Additionally, you can register your phone number on a national do not call registry to reduce the number of telemarketing calls you receive. Also, you can install call-blocking apps on your device, which can help to detect and block fraudulent calls, as well as unwanted telemarketing calls [15].

Fraudulent toll fraud

An attempt is made to trick users into subscribing to or purchasing content through their mobile phone bills.
The term "toll fraud" refers to any type of billing, with the exception of premium SMS and premium calls. These include direct carrier billing, wireless access points (WAPs), and mobile airtime transfers. WAP fraud is one of the most common types of toll fraud. It is possible to commit WAP fraud by tricking the user into clicking a button on a silently loaded, transparent WebView. As soon as the action is performed, an automatic recurring subscription is initiated, and the confirmation SMS or email is hijacked to conceal the financial transaction from the user.

Commercial spyware (stalkerware)

An application that collects and/or transmits sensitive or personal user data without adequate notice or consent and does not display a persistent notification.
By monitoring personal or sensitive user data, stalkerware apps target users and transmit or make this data available to third parties.
Providing that they comply fully with the requirements described below, only surveillance apps designed and marketed specifically for parents to track their children or manage their business are acceptable. Even with the knowledge and consent of another individual (such as a spouse), these apps cannot be used to track them, regardless of whether persistent notifications are displayed consent.

Denial of service (DoS)
Contains code that executes a denial-of-service (DoS) attack or is part of a distributed DoS attack against other systems and resources without the knowledge of the user.
By sending a large number of HTTP requests, for example, remote servers may be subject to excessive load.

Hostile downloaders

An application that is not in itself potentially harmful, but that downloads other potentially harmful applications.
It is possible that the code is a hostile downloader if:

- In addition, there is reason to believe that it was created to spread PHAs, has downloaded PHAs, or contains code that can download and installing apps.
- At least 5% of PHAs are downloaded with a minimum threshold of 500 downloads (25 downloads of PHAs).

Browsers and file-sharing apps are not considered hostile downloaders if:

- Downloads are not driven by them without the involvement of the user.
- Consenting users initiate all PHA downloads.

Non-Android threat

The code contains threats that are not specific to Android.
Android users and devices cannot be harmed by these apps, but they contain components that could be harmful to other platforms.

Phishing

Phishing attacks on Android devices can take the form of fake apps, text messages, or emails that look like they are coming from trusted sources. In these attacks, users are often asked to enter personal information or click on links to a fake website that steals their data. To protect yourself from phishing attacks on your Android device, it is important to be vigilant and skeptical of unsolicited messages and emails. To help you stay safe, here are a few tips:

Be wary of unexpected messages or emails that ask for personal information, such as login credentials or credit card numbers. The information will never be requested by an unsolicited email or message from a legit organization [4].

- Be cautious of links in messages or emails, even if they appear to be from a legitimate source.
- Keep your device and apps updated to ensure that you have the latest security patches and features.
- Use a reputable mobile security app to protect your device from malware and other threats.

- Be aware that phishing scams can also be spread through social media and instant messaging platforms, so be cautious of any unexpected or suspicious messages you receive on these platforms as well.

Elevated privilege abuse

Code that breaks the app sandbox, gains elevated privileges, or disables access to core security-related functions compromises the integrity of the system. The following are examples:

- Apps that violate the Android permissions model or steal credentials (such as OAuth tokens) from other apps.
- The use of features that prevent an application from being uninstalled or stopped.
- SELinux is disabled by this application.

An app that escalates privileges without the consent of the user is classified as a rooting application.

Ransomware

Ransomware is a rapidly growing threat in the cyber world, and Android devices are not immune to these attacks. A ransomware threat is a type of malware that encrypts a victim's files and then demands a ransom payment from the victim in exchange for the decryption key. This type of attack can have severe consequences, as it can lock users out of their own devices and cause them to lose access to important files and data.

Ransomware can spread on Android devices in a variety of ways, including through malicious apps, email attachments, or links in text messages. These attacks often trick users into installing malware on their devices by disguising it as a legitimate app or by using social engineering techniques. Once the malware is installed, it will lock the victim's device and display a ransom message, often demanding payment in the form of cryptocurrency.

To protect yourself from ransomware attacks on your Android device, it is essential to be cautious when downloading apps, opening email attachments, or clicking on links in text messages. Users should only download apps from the official Google Play Store, and should be wary of apps that request unnecessary permissions, such as access to contacts or camera. Additionally, it is important to keep your device and apps updated to ensure that you have the latest security patches and features, as well as to use a reputable mobile security app to protect your device from malware and other threats. Another important step that users can take is to regularly back up their important files and data, as this will allow them to restore their information even if their device is infected with ransomware. This could be done in a cloud service like Google drive, DropBox, iCloud etc. It is important to note that paying the ransom is not a guaranteed solution and is not recommended. Instead, it is crucial to take preventive measures to avoid falling victim to these types of attacks in the first place [16].

Rooting

Rooting code for the device.

Non-malicious rooting code differs from malicious rooting code. As an example, rooting apps inform the user in advance that they intend to root the device, and they do not execute other potentially harmful actions that are applicable to other categories of PHAs. A malicious rooting app doesn't inform the user that it is going to root the device, or it informs the user in advance, but also performs actions that are relevant to other PHA categories.

Spam

Unsolicited messages are sent to the user's contacts or the device is used as an email spam relay.

Spyware

Personal data is transmitted off the device without adequate notice or consent.

The transmission of any of the following information without disclosure or in an unexpected manner is sufficient to constitute spyware:

- Contact list
- The app cannot access photos or other files that are on the SD card or that are not owned by it
- Email content from the user
- Log of calls
- Log of SMS messages
- History of web pages or bookmarks in the default browser
- Other applications' /data/ directories.

Spyware can also be flagged for behavior that can be considered spying on the user. The theft of app data, for example, is a form of recording audio or recording calls made to the phone.
Trojan
A piece of code that appears to be benign, such as a game that claims to be merely a game, but which performs undesirable actions against the user.
In most cases, this classification is used in conjunction with other PHA categories. The trojan consists of an innocuous component and a hidden harmful component. An example of this would be a game that sends premium SMS messages in the background without the user's knowledge.

Uncommon
It is possible for new and rare apps to be classified as uncommon if Google Play Protect does not have enough information to determine whether they are safe or not. This is important to note, however, that this does not imply that the app is necessarily harmful, but it cannot be cleared as safe without further examination.

- Mobile Unwanted Software (MUwS)
- Google defines unwanted software (UwS) as software that is not strictly malicious, but is harmful to the software ecosystem as a whole. The term mobile unwanted software (MUWS) refers to software that impersonates other apps or collects at least one of the following without the consent of the user [5]:
- Device phone number
- Email address for primary correspondence
- Installed applications information
- Third-party account information

III. Mobile Unwanted Software (MUwS)

Unwanted Software Policy and Software Principles offer recommendations for software that provides a great user experience. The Android Unwanted Software Policy expands upon Google's Unwanted Software Policy by establishing Play Store and Android ecosystem principles. As well as providing this information, Android.com also offers this information.

According to the Unwanted Software Policy, most unwanted software exhibits one or more of the following characteristics:
As a result, it is deceptive, as it promises a value proposition that is not met.

- Users are tricked into installing it or it piggybacks on the installation of another program.
- A thorough explanation of all its principal and significant functions is not provided to the user.
- In an unexpected manner, it affects the user's system.
- Private information is collected or transmitted without the knowledge of the user.
- Private information is collected or transmitted without a secure handling (e.g., via HTTPS).

This software is bundled with other software and its presence is not disclosed.
In mobile devices, software takes the form of applications, binary files, framework modifications, etc. Code that violates these principles will be taken down in order to prevent harmful software from entering the software ecosystem or disrupting the user experience.
In the following section, we extend the Unwanted Software Policy to mobile applications. In line with that policy, the Mobile Unwanted Software policy will be revised as needed to address new abuses.

Transparent behavior and clear disclosures

It is imperative that all code fulfills the promises made to the user. Apps should provide all the functionality that has been communicated. Users should not be confused by applications.

- There should be a clear understanding of the functionality and objectives of the app.
- Provide the user with an explicit and clear explanation of what changes will be made to the system by the application. Provide users with the opportunity to review and approve all significant installation options and changes.
- It's vital that software doesn't misrepresent the state of a user's device to them, like when it tells them their system is infected with viruses or in a critical state.
- Do not engage in invalid activities aimed at increasing ad traffic and/or conversions.

- Our policy prohibits the creation of applications that mislead users by impersonating someone else (for example, another developer, company, or entity). Be careful not to imply that your app is associated with or authorized by someone it is not.

Violation examples include:

- Ad fraud
- Impersonation

Protect user data

Whenever possible, users should be able to access, use, collect, and share their sensitive and personal data in a clear and transparent manner.

- You should provide users with the opportunity to approve the collection of their personal data before you begin collecting and sending it from the device. This includes data about third-party accounts, email addresses, phone numbers, installed apps, files, locations, and any other personal and sensitive information that the user would not expect to be collected.
- Users' personal and sensitive information should be handled securely, including by using modern cryptography (for example, HTTPS).
- Generally, software, including mobile apps, should only transmit personal and sensitive information to servers when it is necessary in order to carry out the app's functionality.

Example violations:

- Data Collection (cf Spyware)
- Abuse of restricted permissions

The following are examples of user data policies:

- Google Play User Data Policy
- GMS Requirements User Data Policy
- Google API Service User Data Policy

Do not harm the mobile experience

It is essential that the user experience be straightforward, easy to understand, and based on clear choices made by the user. Providing a clear value proposition to the user should not disrupt the advertised or desired user experience.

- Display ads that do not interfere with the use of a device's functions or display outside the triggering app's environment without adequate consent and attribution to the user.
- It is important for apps not to interfere with other apps or the device's functionality.
- It should be clear how to uninstall, where applicable.
- Mobile software should not mimic the prompts of the device's operating system or other applications. The user should not be suppressed from receiving alerts from other apps or from the operating system, especially those that inform them of changes to their operating system.

Violation examples include:

- Disruptive ads
- Unauthorized Use or Imitation of System Functionality

I.V The categories of Mobile Unwanted Software (MuwS)

Abuse of restricted permissions and data collection

Apps that collect and transmit personal and sensitive information about users without their consent or adequate notice. The list of installed apps, the device phone number, email addresses, location, and other third-party account IDs may be collected.

The use of social engineering

This is an application that pretends to be another app with the intention of deceiving users into performing actions that they intended for the original trusted app.

Disruptive ads

In some cases, advertising is displayed in unexpected ways to users, for example, interfering with the functionality of the device or appearing outside the app's environment without the user's consent or giving them any credit or acknowledgement.
Imitation or unauthorised use of system functionality
Advertisements or applications that mimic or interfere with system functionality, such as notifications or warnings. Notifications at the system level should only be used for the features that are integral to the app.

Imitation or unauthorised use of system functionality

Advertisements or applications that mimic or interfere with system functionality, such as notifications or warnings. Notifications at the system level should only be used for the features that are integral to the app.

Ad fraud

Invalid traffic generated with the intention of tricking an ad network into thinking it comes from an authentic user is classified as ad fraud. The development of ad fraud may result from developers implementing ads in disallowed ways, including hiding ads, automatically clicking ads, altering, or modifying information and otherwise utilizing non-human actions (such as spiders and bots) or human activity designed to generate invalid advertising traffic. Advertisers, developers, and users are adversely affected by invalid traffic and ad fraud, which leads to the erosion of trust in the mobile advertising ecosystem over time.

The following are some examples of common violations:

- An application that renders ads that are not visible to the user.
- Apps that generate clicks on ads without the user's awareness or that generate equivalent network traffic to fraudulently provide click credits.
- Apps that send fake installation attribution clicks to get paid for installations that did not originate from their network.
- An application that displays advertisements when the user is not within the app's interface.
- A false representation of the ad inventory by an app, for example, an app that communicates to ad networks that it is running on an iOS device when, in fact, it is running on an Android device; an app that misrepresents the package name that is being monetized [6].

2. Mobile Malware Techniques

Mobile malware is software that is specifically designed to compromise or damage mobile devices, such as smartphones or tablets. Multiple techniques are commonly employed by mobile malware to infiltrate and compromise devices. These techniques include:

1. Social engineering: This technique involves tricking the user by ensuring that they download and install malware on their device, usually by disguising it as a legitimate app or sending a phishing message.
2. Exploitation of vulnerabilities: Mobile malware can take advantage of vulnerabilities in the device's operating system or in third-party apps to gain access and control of the device.
3. Side loading: This technique involves downloading and installing an app from a source other than an official app store, such as a third-party website. These apps may contain malware that is not detected by the official app store's security checks.
4. Drive-by downloads: This technique involves automatically downloading and installing malware on a device when the user visits a malicious website or clicks on a malicious link.
5. Repackaging: This technique involves taking a legitimate app and adding malware to it before distributing it through official app stores or third-party websites.

To protect against mobile malware, it is important to only download apps from official app stores and to be cautious of any suspicious links or messages. Also, it is important to keep the operating system and apps on the device updated to prevent vulnerabilities.

a. An explanation of the structure of Android packages (APKs)

APKs (Android Packages) are archive files with a .apk suffix that contain all the necessary files (code and assets) to run Android applications.
The command unzip app_name.apk can be used to unpack APK files in Linux [8].

- **METTA-INF:** This file contains the verification information that is generated when an application is signed.
- **MANIFEST.MF:** This file contains a list of names and hashes (usually SHA256 in Base64) for all the files contained within the APK.
- **CERT.SF:** Provides a list of names and hashes of the corresponding lines in MANIFEST.MF.
- **CERT.RSA:** Contains the public key and signature of CERT.SF.
- Assets can include images, videos, documents, databases, etc.

- **lib:** Provides native libraries with compiled code, for a variety of device architectures.
- **res:** A collection of predefined application resources, such as XML files that define color schemes, user interface layouts, fonts, and values.
- **AndroidManifest.xml:** A manifest file that contains information about the application's package name, activities, resources, and version.
- **The classes.dex** contains all the Java classes in the dex file format, which the Android Runtime is able to execute as it is a Dalvik executable that contains all the Java classes.
- The code will be linked to resources using this information.

b. Repackaging

Repackaging is often used to disassemble an application in order to append malicious code to it it should then be reassembled. The reverse engineering process is performed using reverse engineering tools, as shown in the following example.
The malware authors repackage popular Android apps, such as those available on Google Play [6], and distribute them on less monitored and policed third-party app stores. In the process of repackaging, malicious authors alter the signature of the repackaged application, making it appear to be a new application to virus detectors. App repackaging involves the following steps:

- You can download the popular free/paid app from the popular app store(s).
- Using a disassembler such as apktool, disassemble the app.
- The first step is to generate a malicious payload in Java and convert it to bytecode using the dx (part one)
- This tool is part of the Android SDK build tools.
- Add the malware payload to the benign application. The AndroidManifest.xml should be modified
- If necessary, additional resources may be required.
- Using apktool, assemble the modified source again.
- The repackaged app should be distributed by self-signing with another certificate to the lessor monitoring of the third-party app market.

Additionally, repackaging and repackaging techniques can be used to generate a large number of products. There are a number of malware variants. Additionally, it is capable of producing a number of unseen variants. A malware that is already known. In view of the fact that each malware variant has a unique signature. The mutated malware cannot be detected by conventional anti-malware software. Due to the fact that it can damage the reputation of the app distribution market places and pollute them developed by a third party. Additionally, malware writers are able to divert by replacing the original developers' advertisements with new ones, we are able to increase advertisement revenues. In the AndroRAT APK Binder tool [7] repackages and generates a trojanized version of an Android application.

An app that is popular and legitimate that is equipped with Remote Access Trojan functionality (RAT). In other words, the adversary can remotely command the infected device to send SMS messages, obtain the location of the device, record video and/or audio, and access the device's settings. Using the hidden remote access service, you can access device files.

c. Penetration testing tools for Android

That can be used to conduct penetration testing on Android applications. There are some that are used for automated testing, and others that are used for manual testing.

It is beneficial to use automated Android penetration testing tools in order to scan for common vulnerabilities. By combining human intelligence and automated tools, they offer a faster and more cost-effective solution than manual tools and processes.

- The Android Debug Bridge (ADB) is a versatile command-line tool for interacting with Android devices.
- Dex2jar: Converts .dex files to .class files, which are then zipped into a jar file, ready for use.
- JavaD-GUI: This is a standalone graphical utility that allows you to view Java source code directly from the CLASS files.
- Java Dex and APK source code generation using JADX commands and GUIs.
- APKTOOL: Reverse engineering tool for 3rd party, closed, binary Android app compatibility.
- Burp Suite: It can be used to test the security of web applications as part of penetration testing.

- Frida: a tool for reverse engineering, security researchers, and developers.
- The Object: This is a runtime mobile exploration toolkit that will help you assess the security posture of your mobile applications without the need for a jailbreak. The toolkit is powered by Frida and can be used as part of a mobile exploration process.
- Ghidra: A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission.
- Drozer: drozer (formerly Mercury) is the leading security testing framework for Android.
- MobSF: Mobile Security Framework (MobSF) is an automated, all-in-one mobile application (Android/iOS/Windows) pen-testing, malware analysis and security assessment framework capable of performing static and dynamic analysis.

d. Configuring the Android pentesting environment

To set up an Android pentesting environment, you will need to install a few tools on your computer. To produce the best results, the following tools will be need:

1. Android SDK: The Android SDK (Software Development Kit) provides tools for developers to develop Android apps. It includes the Android platform libraries, debugger, emulator, and more. You can download the Android SDK from the Android developer website (https://developer.android.com/sdk).
2. Java Development Kit (JDK): Java Development Kit (JDK) is a set of tools and libraries that are used to develop Java programs. Android app development requires the JDK, so you will need to install it on your system. You can download the JDK from the Oracle website (https://www.oracle.com/java/technologies/javase-downloads.html).
3. Android Studio: It is the official Integrated Development Environment (IDE) for creating and developing Android applications on the Android operating system. It includes all the necessary tools to develop, test, and debug Android apps. You can download Android Studio from the Android developer website (https://developer.android.com/studio).
4. Genymotion: This program lets you run Android apps on your computer via an Android emulator. It is very useful for testing and debugging Android apps. You can download Genymotion from the Genymotion website (https://www.genymotion.com/).

5. Burp Suite: Burp Suite is a toolkit for web application security testing. It includes a proxy server, a web application scanner, and more. You can download Burp Suite from the PortSwigger website (https://portswigger.net/burp).
6. adb: adb (Android Debug Bridge) is a command-line tool that allows you to communicate with an Android device. It is included in the Android SDK and is very useful for debugging and testing Android apps.

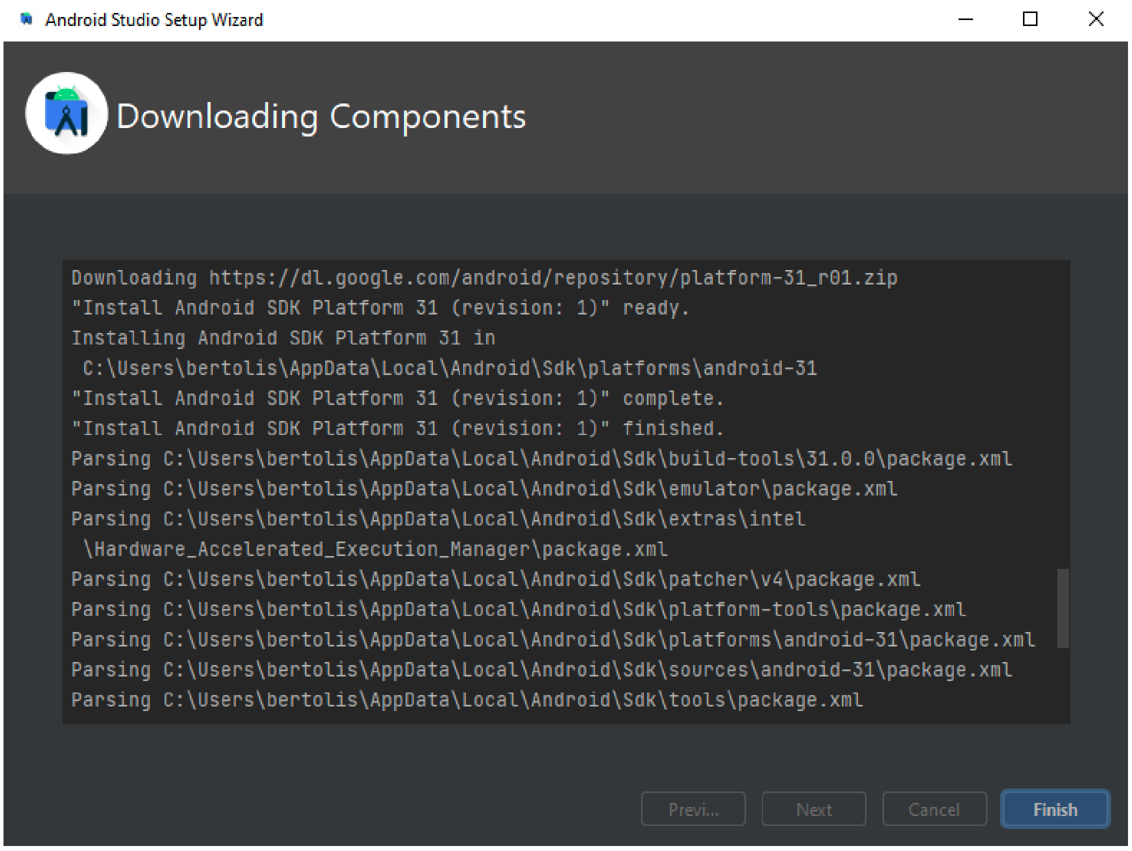
To conduct assessments of Android devices and applications, we require either a real Android device or an emulated Android device. The Android Studio IDE (Integrated Development Environment) provides an Android Virtual Device (AVD), which is a good starting point.

Additionally, Genymotion and Corellium offer cloud-based environments and ARM-based virtualization (the CPU architecture used in mobile devices). With the help of the cloud-based environment, we can spawn and customize mobile devices using the web browser, while Corellium provides the option to root or jailbreak the Android or iPhone device as required.

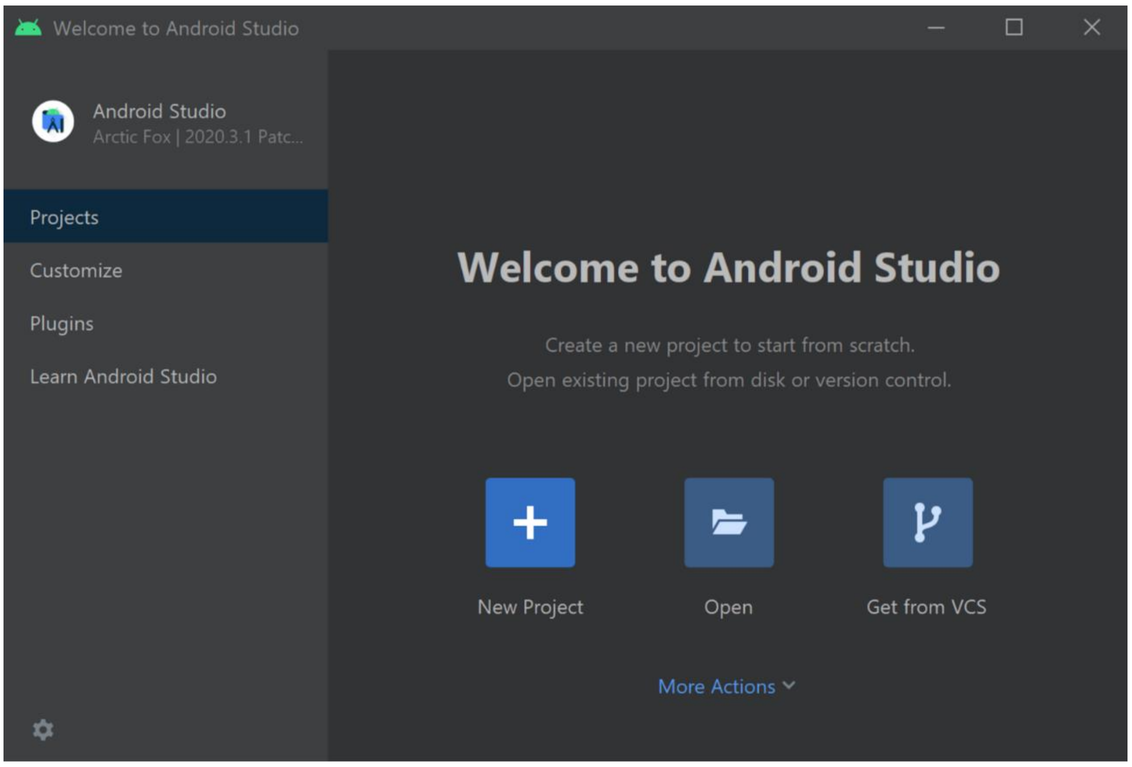
ARM is the CPU architecture used in Android and iPhone devices today. The CPU architecture plays a significant role in kernel exploitation. Due to the fact that most emulators are based on non-ARM CPU architectures, a pentester is not able to work on a potential new kernel exploitation technique while using a mobile emulator. Thanks to ARM-based virtualization, Corellium and Genymotion solve this problem for us.

It is very easy to install Android Studio on Linux. All we need to do is unzip it and run the script studio.sh within the bin/ directory. It is necessary to follow the setup wizard in order to install Android Studio on Windows or macOS. Both operating systems follow a similar process.

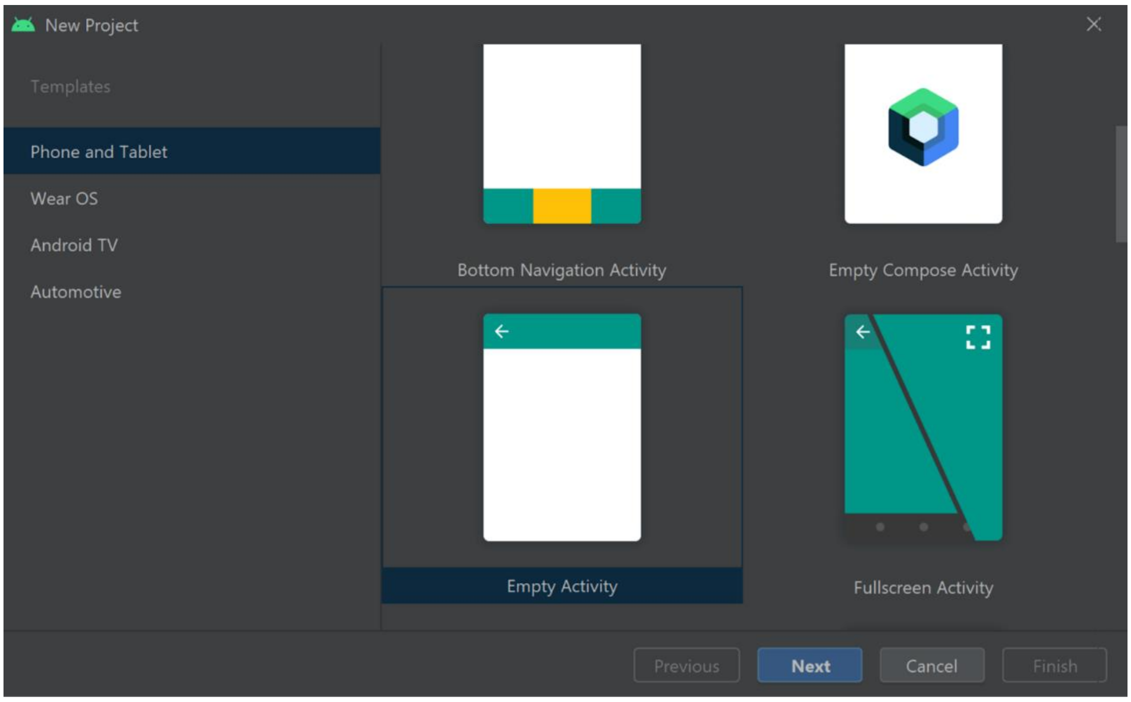
In Windows, for example, we click on the executable and then follow the steps in the setup wizard. Upon completion of the installation, we only need to wait for a few components to download [8].

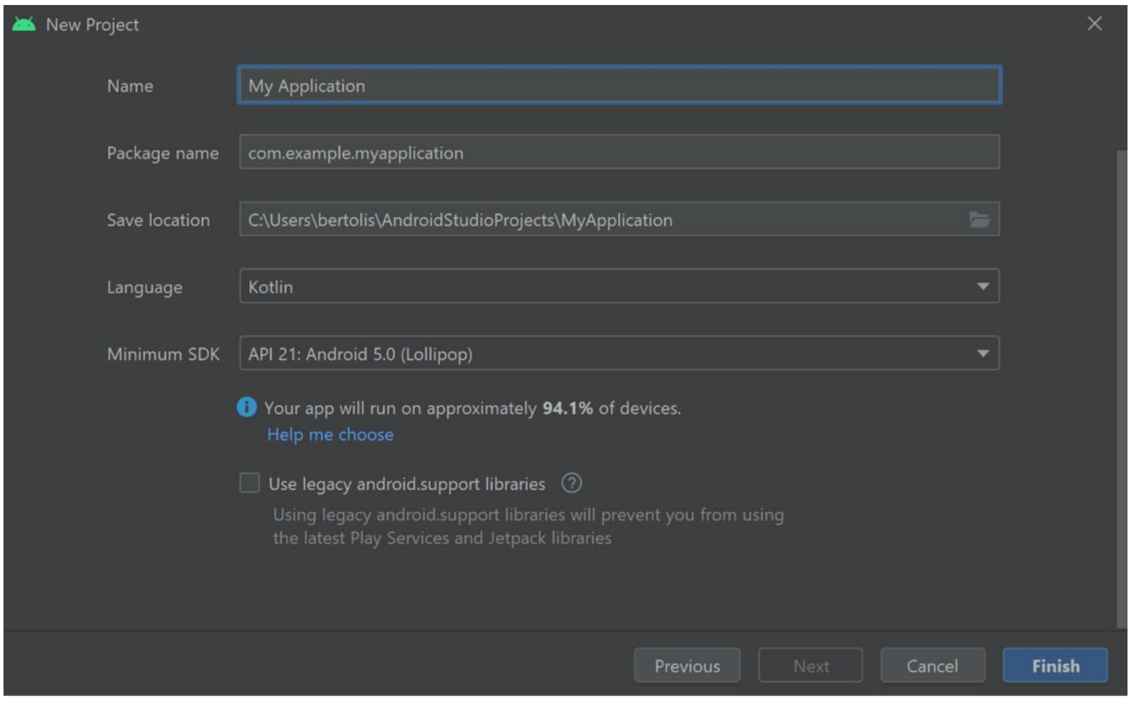


```
Downloading https://dl.google.com/android/repository/platform-31_r01.zip
"Install Android SDK Platform 31 (revision: 1)" ready.
Installing Android SDK Platform 31 in
 C:\Users\bertolis\AppData\Local\Android\Sdk\platforms\android-31
"Install Android SDK Platform 31 (revision: 1)" complete.
"Install Android SDK Platform 31 (revision: 1)" finished.
Parsing C:\Users\bertolis\AppData\Local\Android\Sdk\build-tools\31.0.0\package.xml
Parsing C:\Users\bertolis\AppData\Local\Android\Sdk\emulator\package.xml
Parsing C:\Users\bertolis\AppData\Local\Android\Sdk\extras\intel
 \Hardware_Accelerated_Execution_Manager\package.xml
Parsing C:\Users\bertolis\AppData\Local\Android\Sdk\patcher\v4\package.xml
Parsing C:\Users\bertolis\AppData\Local\Android\Sdk\platform-tools\package.xml
Parsing C:\Users\bertolis\AppData\Local\Android\Sdk\platforms\android-31\package.xml
Parsing C:\Users\bertolis\AppData\Local\Android\Sdk\sources\android-31\package.xml
Parsing C:\Users\bertolis\AppData\Local\Android\Sdk\tools\package.xml
```



Create a new project to start from scratch.
Open existing project from disk or version control.





Name: My Application

Package name: com.example.myapplication

Save location: C:\Users\bertolis\AndroidStudioProjects\MyApplication

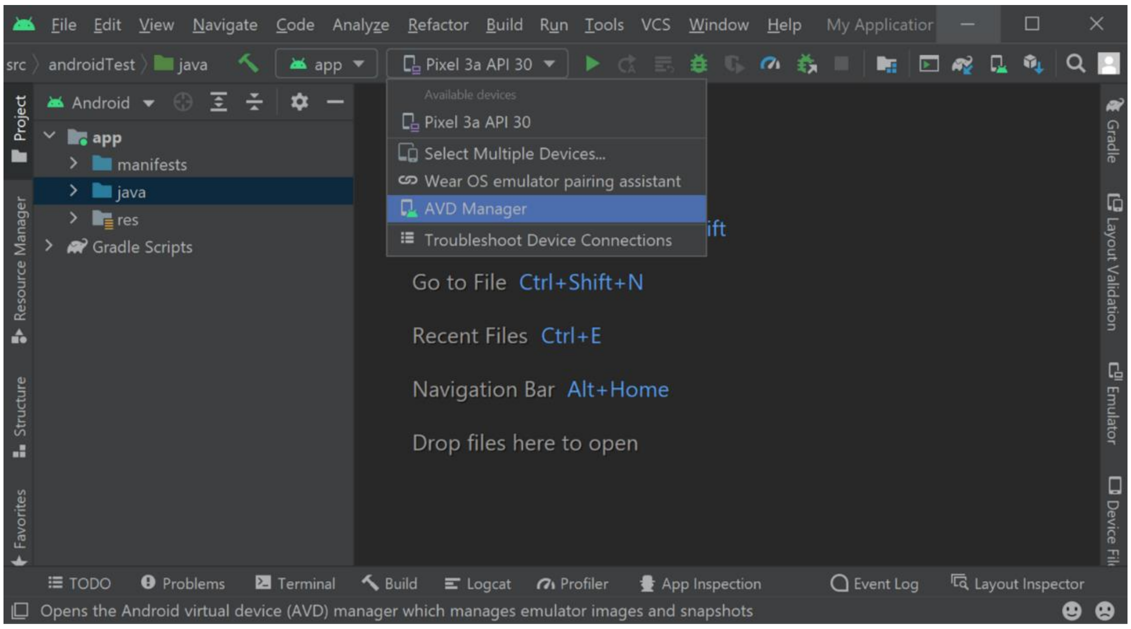
Language: Kotlin

Minimum SDK: API 21: Android 5.0 (Lollipop)

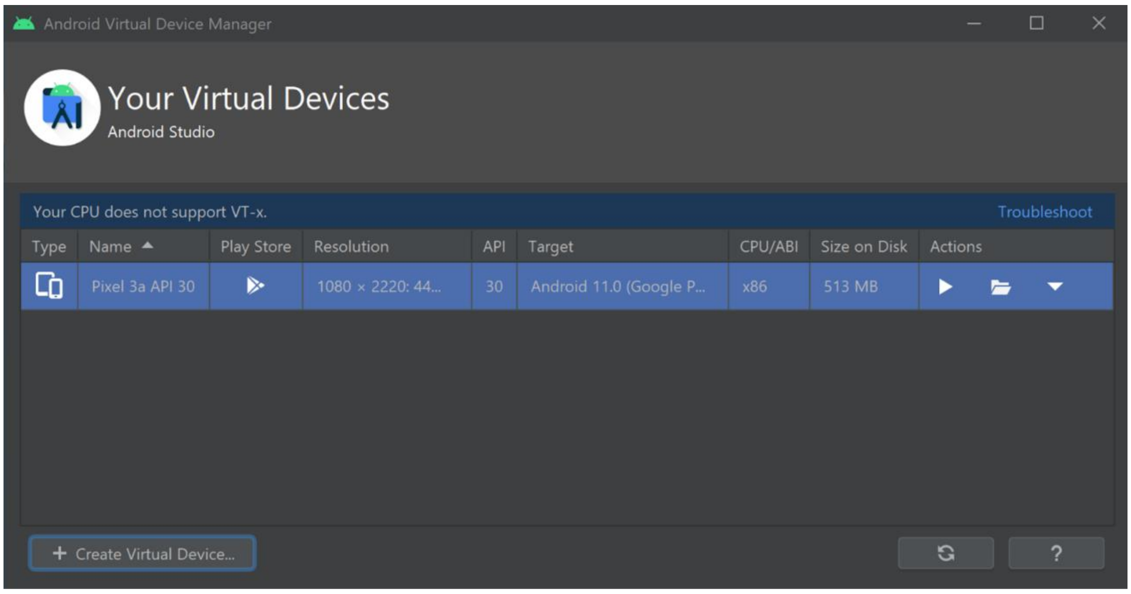
Your app will run on approximately **94.1%** of devices.
Help me choose

☐ Use legacy android.support libraries

Using legacy android.support libraries will prevent you from using the latest Play Services and Jetpack libraries

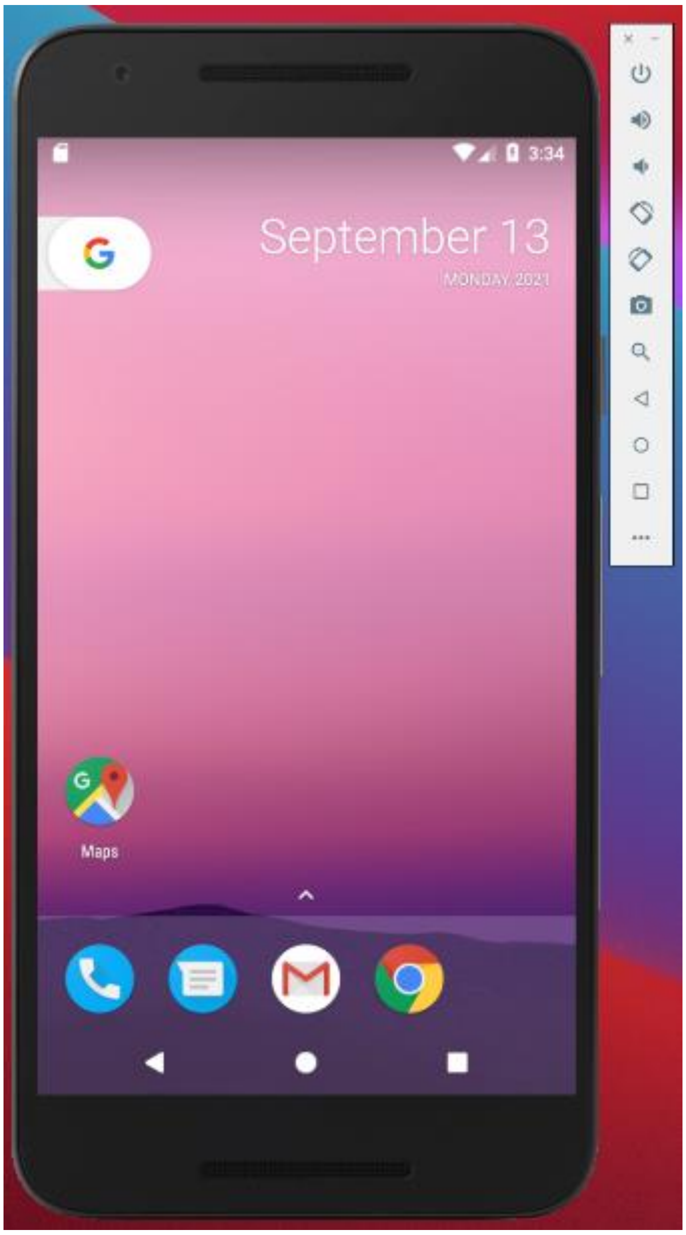

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help My Application
src androidTest java app Pixel 3a API 30
Android
app
manifests
java
res
Gradle Scripts
Available devices
Pixel 3a API 30
Select Multiple Devices...
Wear OS emulator pairing assistant
AVD Manager
Troubleshoot Device Connections
Go to File Ctrl+Shift+N
Recent Files Ctrl+E
Navigation Bar Alt+Home
Drop files here to open
TODO Problems Terminal Build Logcat Profiler App Inspection Event Log Layout Inspector
Project Resource Manager Structure Favorites
Gradle Layout Validation Emulator Device File

Opens the Android virtual device (AVD) manager which manages emulator images and snapshots



Your CPU does not support VT-x.

Type	Name	Play Store	Resolution	API	Target	CPU/ABI	Size on Disk	Actions
	Pixel 3a API 30		1080 × 2220: 44...	30	Android 11.0 (Google P...	x86	513 MB	


3:34
September 13
MONDAY, 2021
Maps

3. Pentesting Android Apps Using Frida

a. Introduction to Frida

No Android application review is complete without reverse engineering the app to find out what's running in the background. A tool called Frida is used in this essay to dynamically modify the behavior of the app during runtime.

In situations where certain sensitive functionalities in an Android application, such as fingerprint authentication, are disabled or not permitted to run on rooted phones, or if you wish to bypass a login screen or disable SSL certificate pining so that traffic can be intercepted, it is necessary to modify the behavior of an Android application.

It has traditionally been the case that if anyone wished to modify a particular functionality, they would need to use one of the following methods:

1. Edit and repackage the decompiled Smali files, which can be challenging since it can be difficult for beginners to understand the code that has been decompiled, especially in the beginning.
2. Today, pentesters are more likely to use the Xposed Framework, which requires them to rewrite the functionality in a separate app and relaunch it

Frida allows you to modify an application's code without relaunching or repackaging it, unlike the previous two approaches.

As described by its creators, Frida is a toolkit for dynamic code instrumentation. *Injecting JavaScript or your own library into native apps can be done on Windows, macOS, Linux, iOS, Android, and QNX.*

For a better understanding of this, let us consider a LoginActivity that runs on your Android application and waits for the user name and password to be entered. In this activity, there is a function defined called "checkLogin" that returns a true or false value depending on the validity of the credentials. By overriding this "checkLogin" function in memory with the code that we have written, Frida allows us to modify the functionality dynamically.

In this post, we will demonstrate the power of Frida by utilizing two purposefully built vulnerable Android applications. Bypassing the login screen and the root detection logic.

1. MWR Labs has developed a vulnerable password storage application called Sieve
2. Black Hat 2015-2016 Arsenal: InsecureBankv2 - A vulnerable banking application

We recommend that you have a working Android device, a Python environment, as well as a rooted Android phone with an ARM architecture before proceeding with the setup. As of this writing, we were unable to get Frida running satisfactorily on an unrooted Android device. Frida is currently supporting only ARM architectures [9].

b. Setting up Frida

Frida consists of two components, namely. Client and server communicate over two ports (TCP) 27042 and 27043. The client can be installed by executing our favourite pip command.
Please note that this installation requires administrator/root privileges on your Windows/Unix computer. Make sure you run the following command using sudo or a command shell with administrator privileges.
pip install frida
In order to identify what version of Frida client you have installed on your machine, start your command prompt and enter the following command.
frida –version
In order to install the server, you must browse the releases directory and download the file corresponding to your mobile device's platform and the version shown above. If the version is incorrect, it will not work.
You will need to unzip the file on your mobile device and place it in a folder of your choice, preferably the /data/local/tmp folder of your mobile device.
Using the command below, modify the permissions for the frida-server binary and run it as shown.
On your desktop, run the following command to test the connection with the frida-server.
frida-ps -aU
Basically, the output shows all currently running injectable processes [9].

We are now ready to begin using Frida for our assessment.

I. Login Bypass Using Frida

Here we will take a look at a practical demonstration of how a login bypass can be implemented in the Sieve app. You will need to open the Sieve app on your Android device.
As soon as we enter a valid set of credentials, the application opens the main screen that shows us all of the passwords that we have saved with the program.
You can identify the sieve process running on your computer by running the following command at a command prompt.

frida-ps -aU | grep -i "sieve"

It would be helpful to decompile the sieve app before we read about the login bypass, so that we can understand how the login functionality is working. As a first step, you will need to decompile the apk using the dex2jar tool and then view the final jar file in the JD-GUI after you have decompiled the apk.
We were able to find a function named "checkKeyResult" within the decompiled APK, which can be found in the decompiled APK. The application redirects the flow to the loginSuccessful() function if the boolean value is true, otherwise it redirects it to the loginFailed() function if the boolean value is false [9].
With Frida, you can create bindings with a variety of languages such as Python, C, .NET, and Swift. It should be noted, however, that we will be using Python to demonstrate the bypass. If we take a closer look at this little piece of code, we can get a better understanding of what Frida is doing by taking a look at it in detail. We are merely overriding our target function "checkKeyResult" by inserting the boolean value as "true”. Here, we are not changing the function itself, but we do have control over what is being passed to the function and what is being returned from the function.
The lines 25-26 are used to attach to our target process which we found by using the search function in the system.

frida-ps -aU command

From line 10-23, you will find the code written in Javascript that needs to be executed in a process through Frida. The application needs to be started on the device and then the

Python script needs to be executed. Upon executing the script, it will wait for the "checkKeyResult" function to be called once it has completed executing the script.

After clicking the Sign In button, the "checkKeyResult" function will be called in the runtime, and Frida will inject our JS into the runtime, passing a true value to the function, and after that we will be able to bypass the login process completely.
If you push the Sieve app into the background then try to bring it back when you're ready, it will ask for a pin if you've pushed the app into the background and then tried to bring it back. The below script can be used in order to bruteforce this pin in order to gain access to the application by bruteforcing it.

II. Bypass Root Detection

In Android app tests, bypassing root detection is one of the most important use cases that can be tested. Some applications check for rooted devices during installation or restrict the use of sensitive functionalities such as fingerprint authentication. It is possible to detect root privileges by checking for installation of most common APKs such as SuperSu, which governs the root privileges. It is also possible to detect root privileges by trying to write into the protected directories of the Android file system like root.
The APK must be decompiled, the smali files must be edited, and then you must repackage the APK by patching the methods that are implemented for root detection so that they are bypassed. While it's a fairly cumbersome process, it can be made relatively simple by using Frida as we'll see in the following example of InsecureBank2 Android application, which we'll use as an example.
The InsecureBank2 application displays a warning message to the user upon successful login to the application informing them that their device has been rooted.

To determine whether the device is rooted, two functions are used by the InsecureBank app, as shown below in the decompiled APK.

- doesSUexist
- doesSuperuserApkExist

Using the javascript code below, we bypass the root detection logic and this is why we receive the message “Device not Rooted”.

The first step is to run this script and then press the Sign In button in the app, as shown in the screenshot below.

The root detection logic is invoked once the Sign In button has been pressed and the PostLogin Activity has been initiated. Frida bypasses the detection successfully since it uses our JS code, as shown below.

4. Android Security Tools: Burp-Suite, Run Frida Server, Install-Uninstall apk

Burp Suite is a toolkit for web application security testing. It provides a set of tools that are used to do a wide range of tasks, including intercepting traffic, manipulating requests and responses, and scanning for vulnerabilities.
To use Burp Suite, you will need to install it on your computer and then configure your Android device to use it as a proxy. This can be done by going to Settings > Network & internet > Wi-Fi > [Wi-Fi network] > Advanced > Private DNS and setting it to "localhost".
Frida is a dynamic instrumentation tool that can be used to hook into and modify the behavior of applications. To use Frida with an Android device, you will need to install the Frida server on the device and then connect to it from your computer.
To install the Frida server on an Android device, you will need to have a rooted device and then follow these steps:

1. Download the Frida server APK file from the Frida website.
2. Transfer the APK file to the Android device.
3. On the Android device, navigate to the APK file and install it.

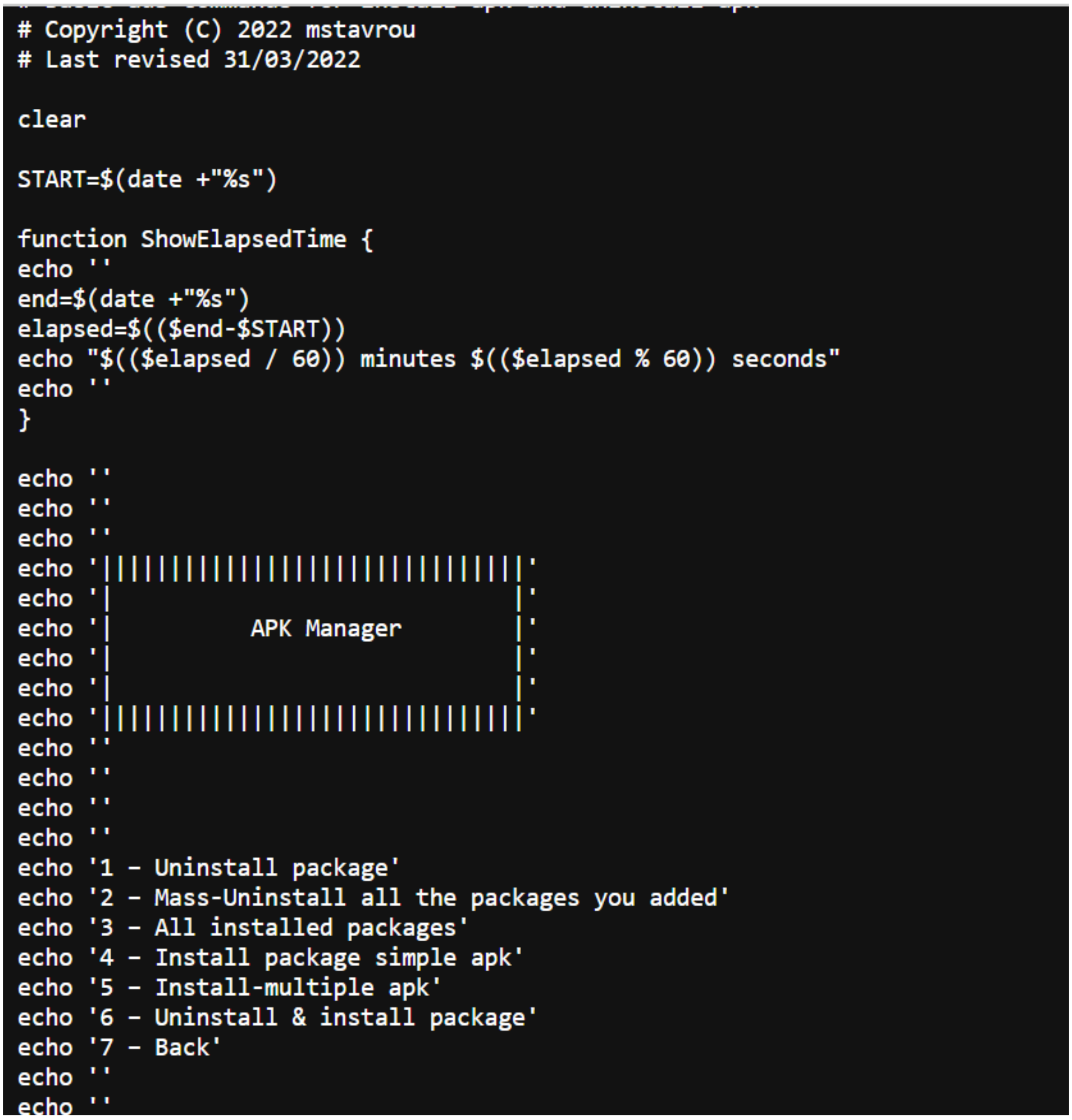
To uninstall an APK from an Android device, follow these steps:

1. Go to Settings > Apps & notifications > [app name] > Uninstall.

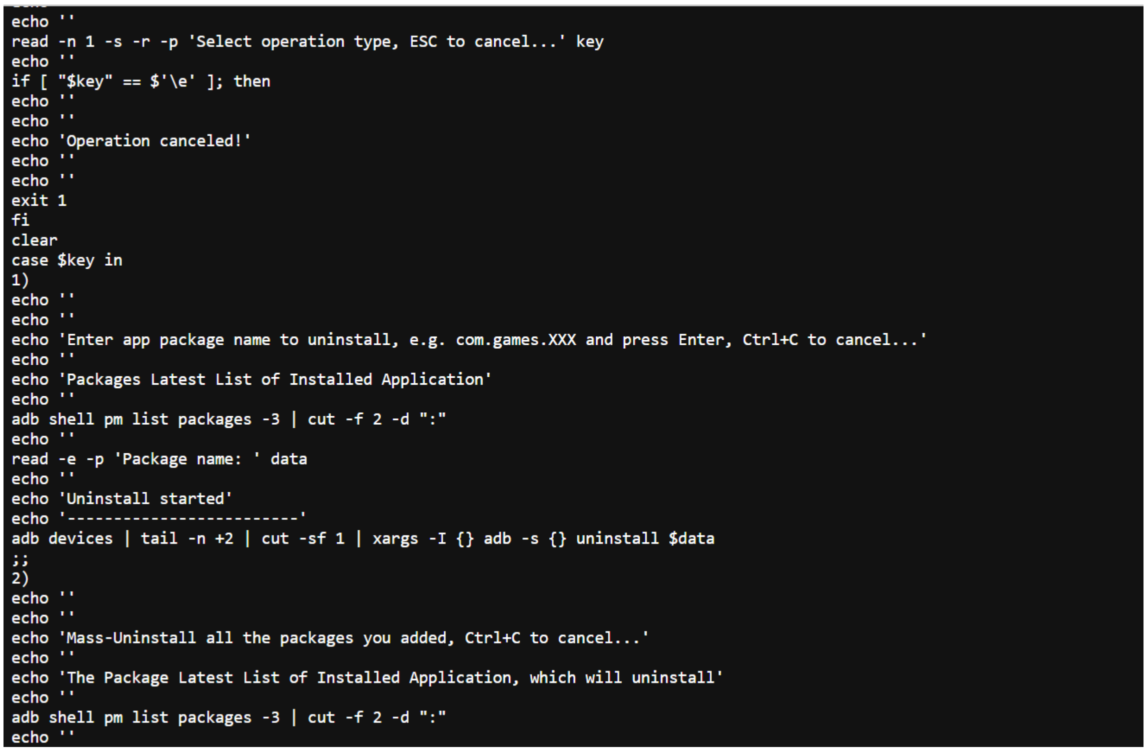
2. Tap "OK" to confirm.

Keep in mind that uninstalling an app will delete all of its data, including any files or settings that it has created.

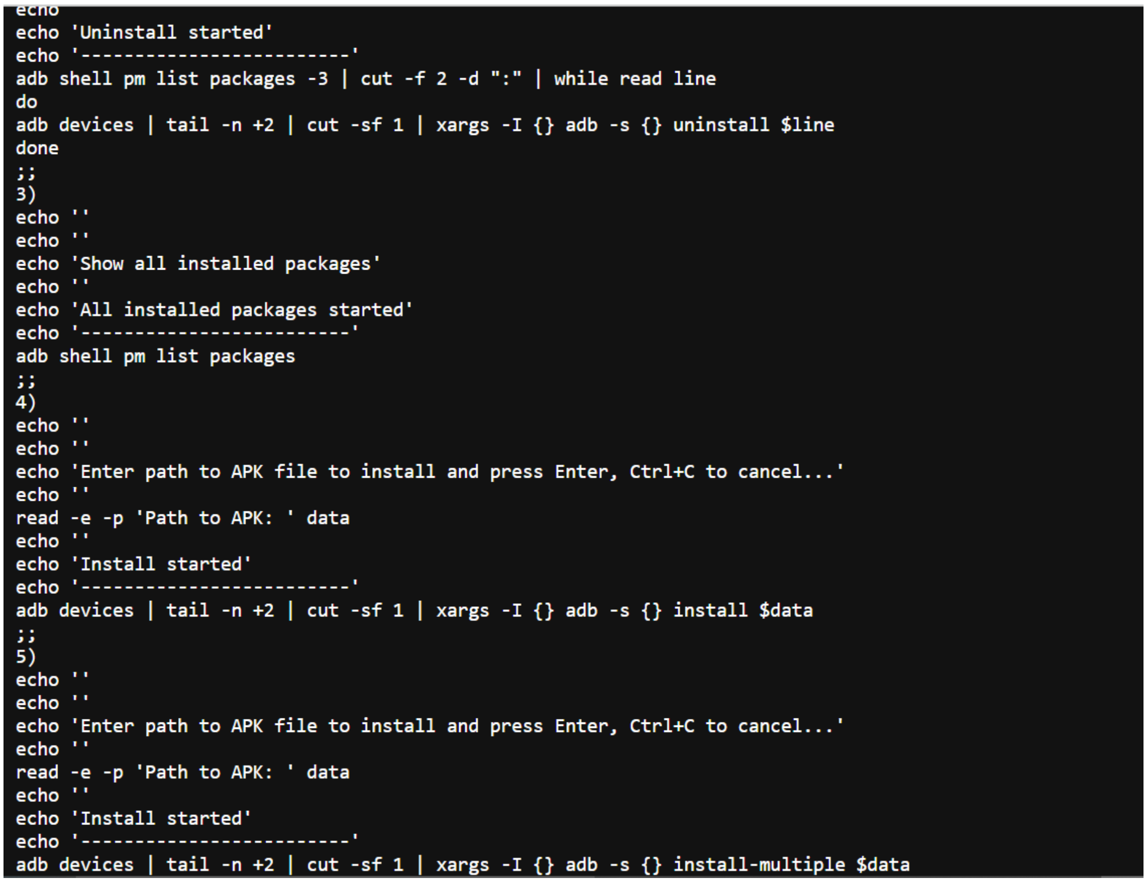
a. Introduction of security develop tools



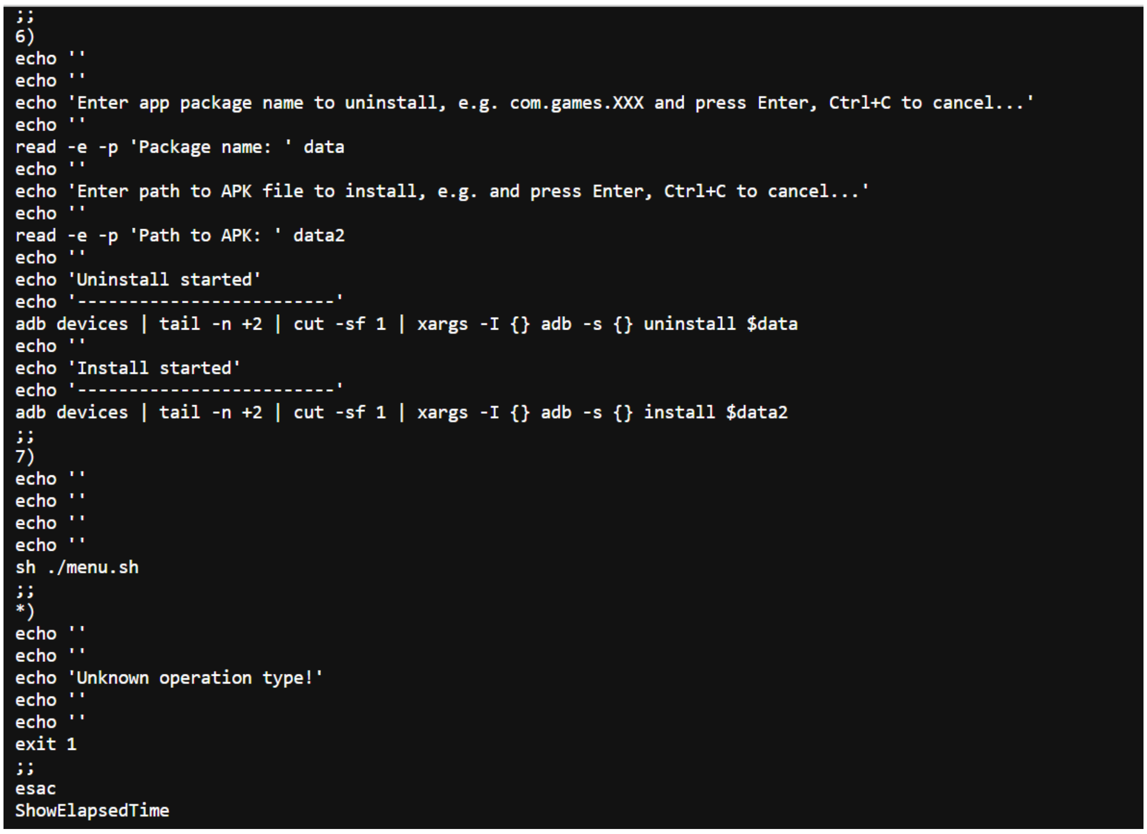
```
# Copyright (C) 2022 mstavrou
# Last revised 31/03/2022

clear

START=$(date +"%s")

function ShowElapsedTime {
echo ''
end=$(date +"%s")
elapsed=$(($end-$START))
echo "$(($elapsed / 60)) minutes $(($elapsed % 60)) seconds"
echo ''
}

echo ''
echo ''
echo ''
echo '||||||||||||||||||||||||||||||'
echo '|                            |'
echo '|         APK Manager        |'
echo '|                            |'
echo '|                            |'
echo '||||||||||||||||||||||||||||||'
echo ''
echo ''
echo ''
echo ''
echo '1 - Uninstall package'
echo '2 - Mass-Uninstall all the packages you added'
echo '3 - All installed packages'
echo '4 - Install package simple apk'
echo '5 - Install-multiple apk'
echo '6 - Uninstall & install package'
echo '7 - Back'
echo ''
echo ''
```



```
echo ''
read -n 1 -s -r -p 'Select operation type, ESC to cancel...' key
echo ''
if [ "$key" == $'\e' ]; then
echo ''
echo ''
echo 'Operation canceled!'
echo ''
echo ''
exit 1
fi
clear
case $key in
1)
echo ''
echo ''
echo 'Enter app package name to uninstall, e.g. com.games.XXX and press Enter, Ctrl+C to cancel...'
echo ''
echo 'Packages Latest List of Installed Application'
echo ''
adb shell pm list packages -3 | cut -f 2 -d ":"
echo ''
read -e -p 'Package name: ' data
echo ''
echo 'Uninstall started'
echo '-------------------------'
adb devices | tail -n +2 | cut -sf 1 | xargs -I {} adb -s {} uninstall $data
;;
2)
echo ''
echo ''
echo 'Mass-Uninstall all the packages you added, Ctrl+C to cancel...'
echo ''
echo 'The Package Latest List of Installed Application, which will uninstall'
echo ''
adb shell pm list packages -3 | cut -f 2 -d ":"
echo ''
```



```
echo
echo 'Uninstall started'
echo '-------------------------'
adb shell pm list packages -3 | cut -f 2 -d ":" | while read line
do
adb devices | tail -n +2 | cut -sf 1 | xargs -I {} adb -s {} uninstall $line
done
;;
3)
echo ''
echo ''
echo 'Show all installed packages'
echo ''
echo 'All installed packages started'
echo '-------------------------'
adb shell pm list packages
;;
4)
echo ''
echo ''
echo 'Enter path to APK file to install and press Enter, Ctrl+C to cancel...'
echo ''
read -e -p 'Path to APK: ' data
echo ''
echo 'Install started'
echo '-------------------------'
adb devices | tail -n +2 | cut -sf 1 | xargs -I {} adb -s {} install $data
;;
5)
echo ''
echo ''
echo 'Enter path to APK file to install and press Enter, Ctrl+C to cancel...'
echo ''
read -e -p 'Path to APK: ' data
echo ''
echo 'Install started'
echo '-------------------------'
adb devices | tail -n +2 | cut -sf 1 | xargs -I {} adb -s {} install-multiple $data
```



```
;;
6)
echo ''
echo ''
echo 'Enter app package name to uninstall, e.g. com.games.XXX and press Enter, Ctrl+C to cancel...'
echo ''
read -e -p 'Package name: ' data
echo ''
echo 'Enter path to APK file to install, e.g. and press Enter, Ctrl+C to cancel...'
echo ''
read -e -p 'Path to APK: ' data2
echo ''
echo 'Uninstall started'
echo '------------------------'
adb devices | tail -n +2 | cut -sf 1 | xargs -I {} adb -s {} uninstall $data
echo ''
echo 'Install started'
echo '------------------------'
adb devices | tail -n +2 | cut -sf 1 | xargs -I {} adb -s {} install $data2
;;
7)
echo ''
echo ''
echo ''
echo ''
sh ./menu.sh
;;
*)
echo ''
echo ''
echo 'Unknown operation type!'
echo ''
echo ''
exit 1
;;
esac
ShowElapsedTime
```

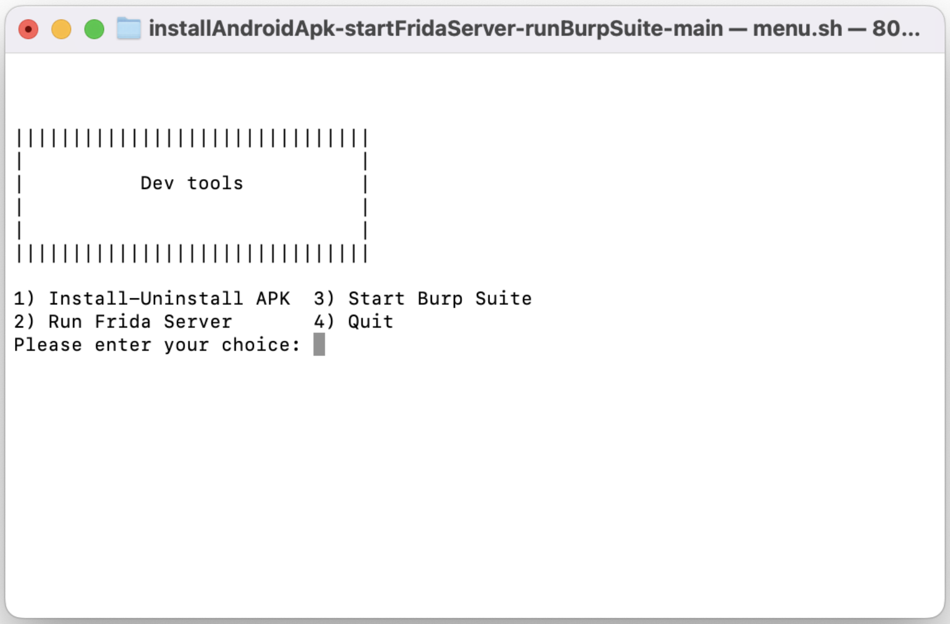
This script is a Bash script that utilizes the Android Debug Bridge (adb) command-line tool to perform various actions related to Android applications. The script starts by displaying a menu with several options for the user to choose from, including options to uninstall a specific package, mass-uninstall all added packages, display all installed packages, install a single package, install multiple packages, uninstall and install a package, and exit the script.

The script uses adb commands to perform these actions. For example, when the user selects the option to uninstall a specific package, the script prompts the user to enter the package name and then uses the adb command "adb uninstall" followed by the package name to uninstall the package. Similarly, when the user selects the option to display all installed packages, the script uses the adb command "adb shell pm list packages" to display all installed packages.

One interesting aspect of the script is its use of the "adb devices" command, which is used to list all connected devices. The script then uses the "xargs" command to run adb commands on all connected devices, rather than just a single device. This allows the script to perform the same action on multiple devices at once.

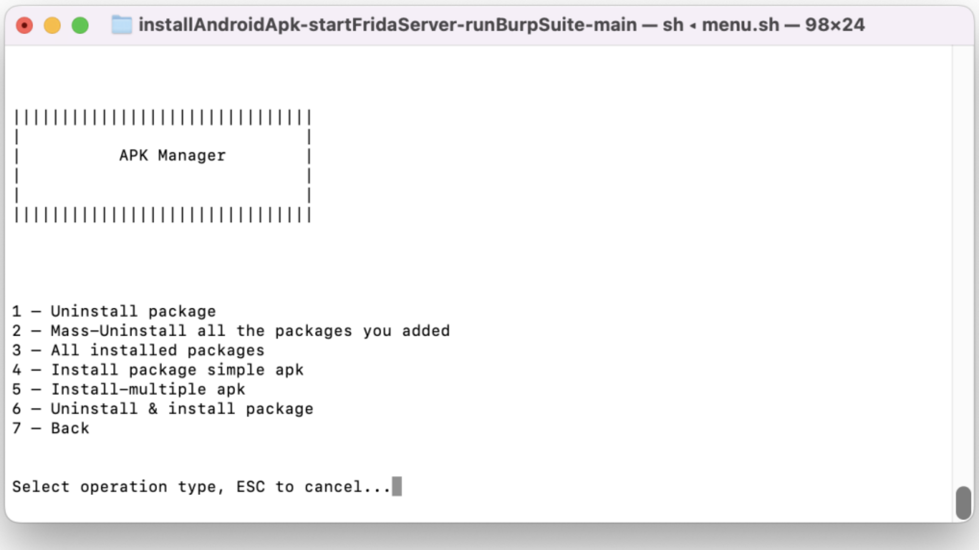
Additionally, the script also includes a function that calculates and displays the elapsed time since the script was started. This can be useful to understand how long certain operations take and to monitor the script performance.
Overall, this script provides a useful tool for managing Android applications and it utilizes various adb commands in a user-friendly manner. As a result, repetitive tasks can be automated and also perform actions on multiple connected devices which can be helpful for developers or testers.
There are some instructions that need to be followed first in order to be able to run the script tool successfully.

1. Rename the frida server to 'fridaServer' in android emulator ex. /data/local/tmp/fridaServer
2. The second step is to run the command: chmod +x menu.sh
3. Finally, run the following command: ./menu



installAndroidApk-startFridaServer-runBurpSuite-main — menu.sh — 80...

```
||||||||||||||||||||||||||||||
|                            |
|          Dev tools         |
|                            |
|                            |
||||||||||||||||||||||||||||||

1) Install-Uninstall APK  3) Start Burp Suite
2) Run Frida Server       4) Quit
Please enter your choice: 
```


installAndroidApk-startFridaServer-runBurpSuite-main — sh ◂ menu.sh — 98×24

```
||||||||||||||||||||||||||||||
|                            |
|         APK Manager        |
|                            |
|                            |
||||||||||||||||||||||||||||||



1 - Uninstall package
2 - Mass-Uninstall all the packages you added
3 - All installed packages
4 - Install package simple apk
5 - Install-multiple apk
6 - Uninstall & install package
7 - Back

Select operation type, ESC to cancel...
```

When the installation process is complete, you can choose the option you like. You can automatically install or uninstall apk from the android emulator, you can start burp suite, or you can start Frida server depending on your preferences. You can download the develop security tools from my repository in github https://github.com/mathias82/installAndroidApk-startFridaServer-runBurpSuite.

5. Frida Commands

a. Introduction to frida commands and how to use it

The Frida toolkit is a dynamic instrumentation toolkit that lets you inject code into running programs across Windows, macOS, Linux, iOS, Android, and QNX platforms. The program is often used for reverse engineering and dynamic analysis of software as well as for reverse engineering.

In order to get started with Frida, here are some basic commands you can use:

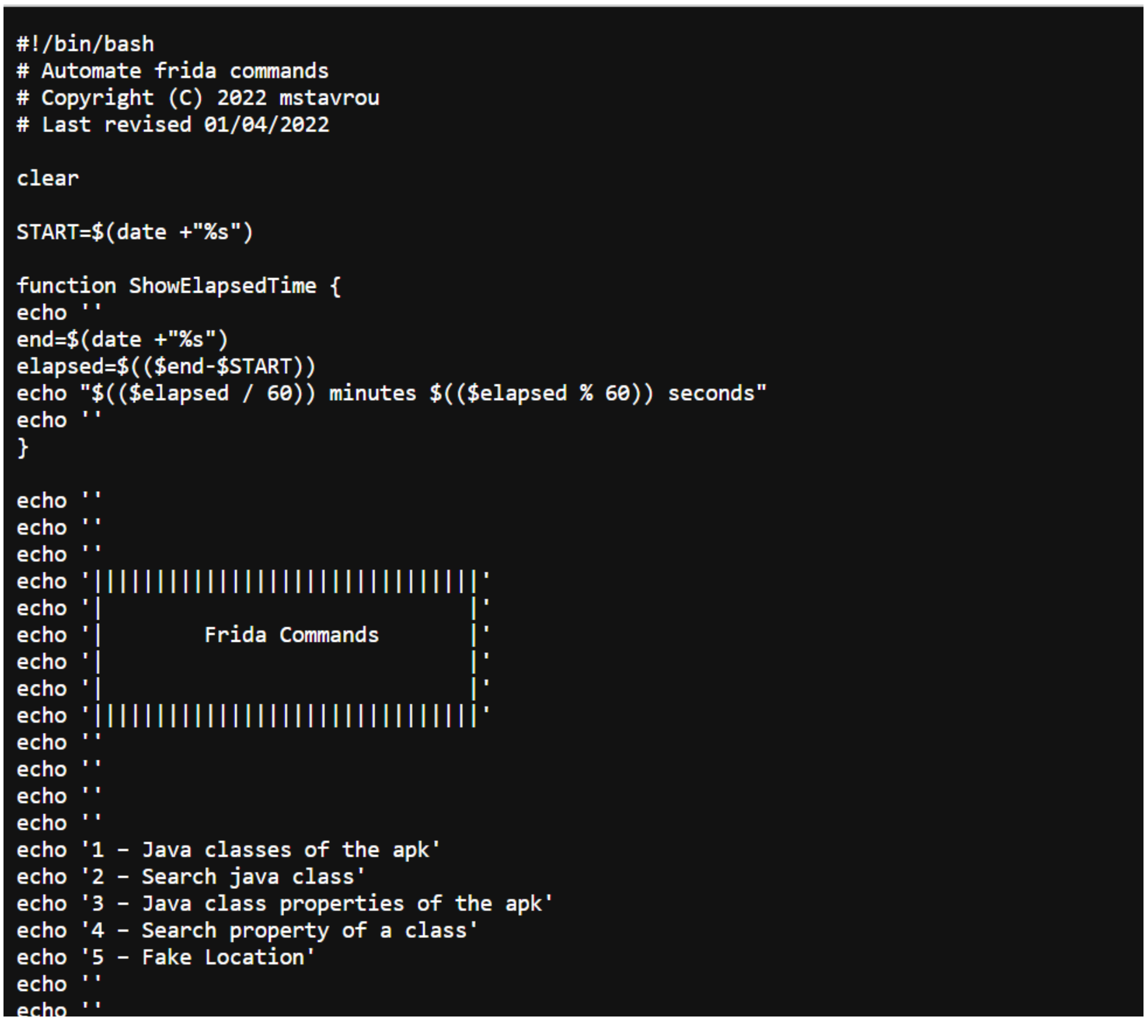
1. You can use the ps command to list all of the processes that are currently running on your device or emulator:
   frida-ps
2. To attach Frida to a process, use the `attach` command. For example, to attach to the process with the ID 12345, use the following command:
   frida -U -f com.example.app -p 12345

3. To run a script with Frida, use the `-l` flag followed by the path to the script. For example: ***frida -U -l script.js -f com.example.app -p 12345***
4. To execute Frida scripts in the Python interpreter, use the frida.get_device_manager() function to get a handle to the device manager, and then use the enumerate_devices() function to get a list of connected devices. Then, use the attach() function to attach Frida to a specific process:

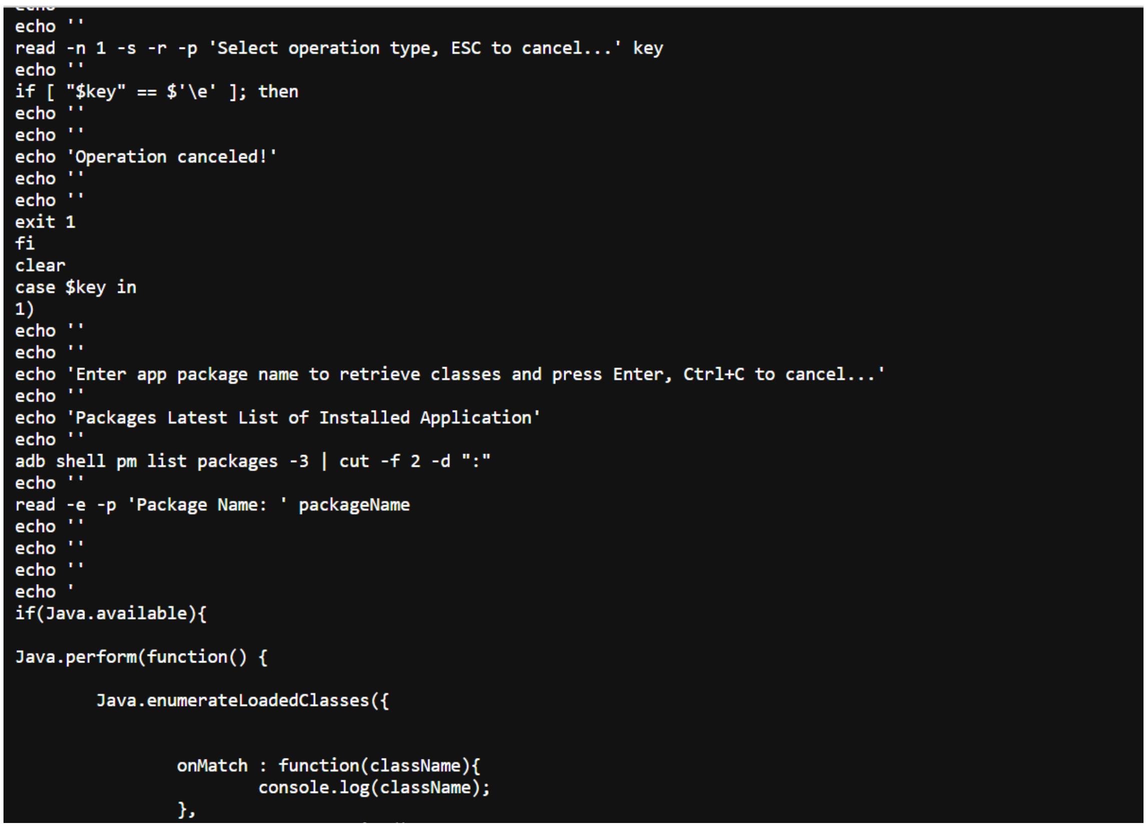
```
import frida

device_manager = frida.get_device_manager()
devices = device_manager.enumerate_devices()

for device in devices:
    print(device)

device = devices[0]
pid = device.spawn(["com.example.app"])
device.resume(pid)
session = device.attach(pid)
```

In addition, there is a script that I have created that has a menu of develop tools. The script contains frida commands, for example such as finding and saving the files of the classes of a package to a document, or finding the methods of a class and saving them to the output-fridaTxt folder. Basically, this program creates a javascript document and stores it in the output directory. As a prerequisite for using the Android emulator, you should have installed and run the Frida server first, in order to use it.

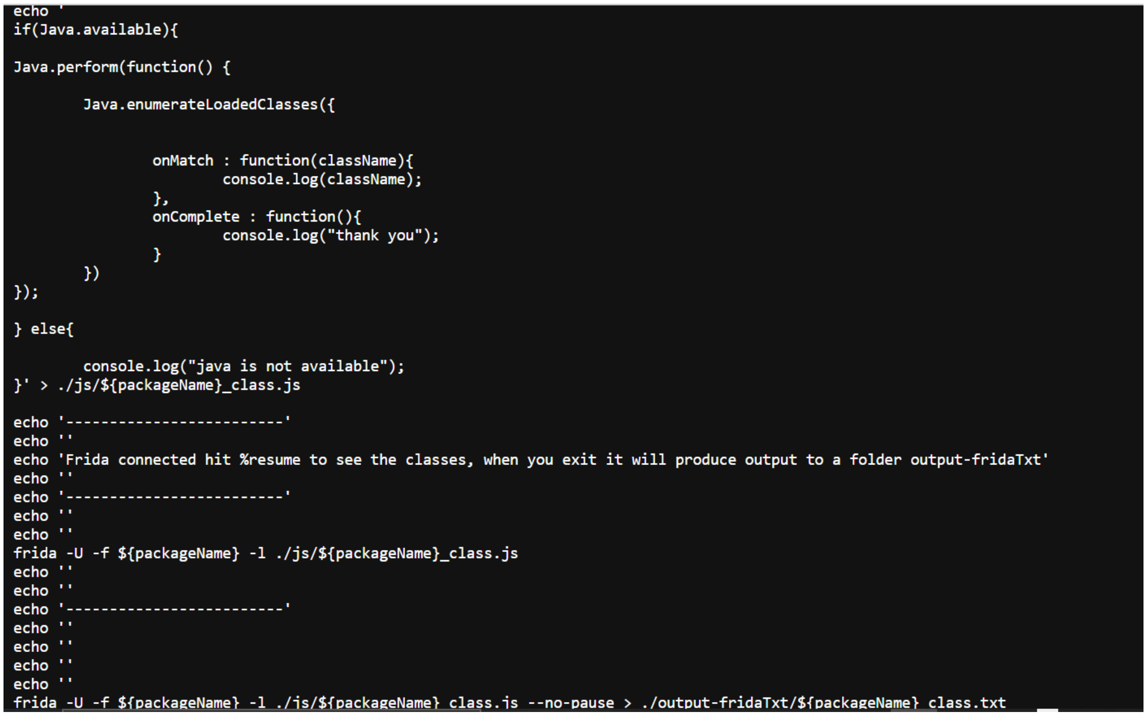
b. Instructions to install automated tool Frida commands



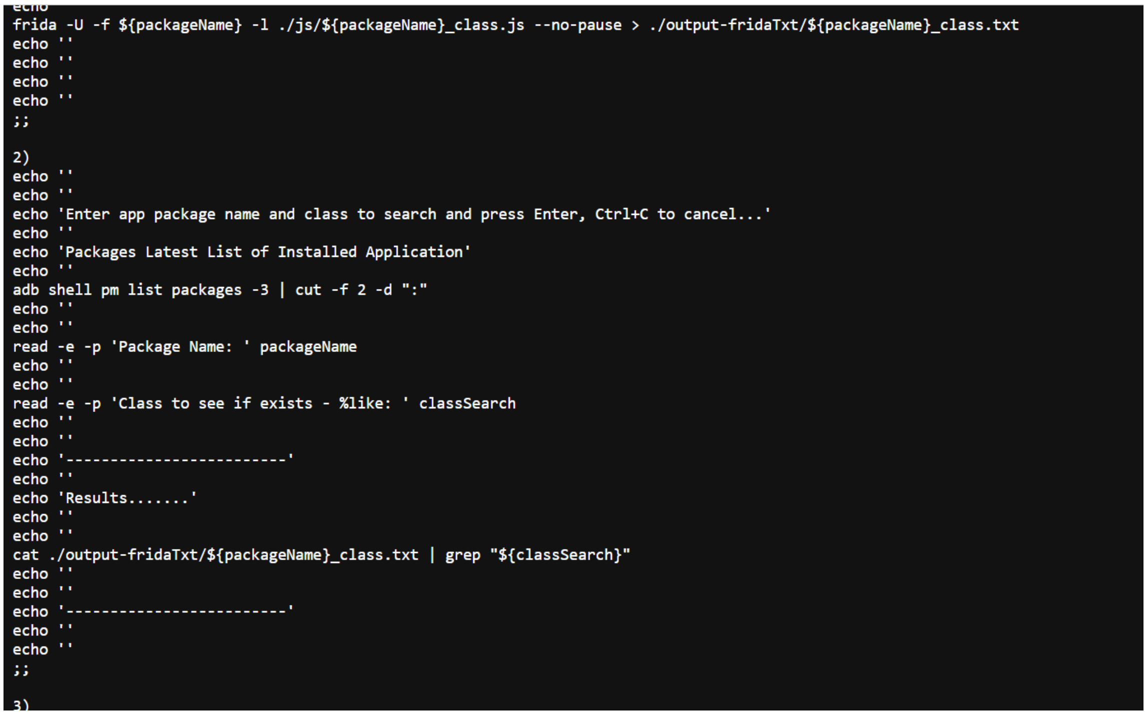
```
#!/bin/bash
# Automate frida commands
# Copyright (C) 2022 mstavrou
# Last revised 01/04/2022

clear

START=$(date +"%s")

function ShowElapsedTime {
echo ''
end=$(date +"%s")
elapsed=$(($end-$START))
echo "$(($elapsed / 60)) minutes $(($elapsed % 60)) seconds"
echo ''
}

echo ''
echo ''
echo ''
echo '|||||||||||||||||||||||||||||||'
echo '|                             |'
echo '|        Frida Commands       |'
echo '|                             |'
echo '|                             |'
echo '|||||||||||||||||||||||||||||||'
echo ''
echo ''
echo ''
echo ''
echo '1 - Java classes of the apk'
echo '2 - Search java class'
echo '3 - Java class properties of the apk'
echo '4 - Search property of a class'
echo '5 - Fake Location'
echo ''
echo ''
```



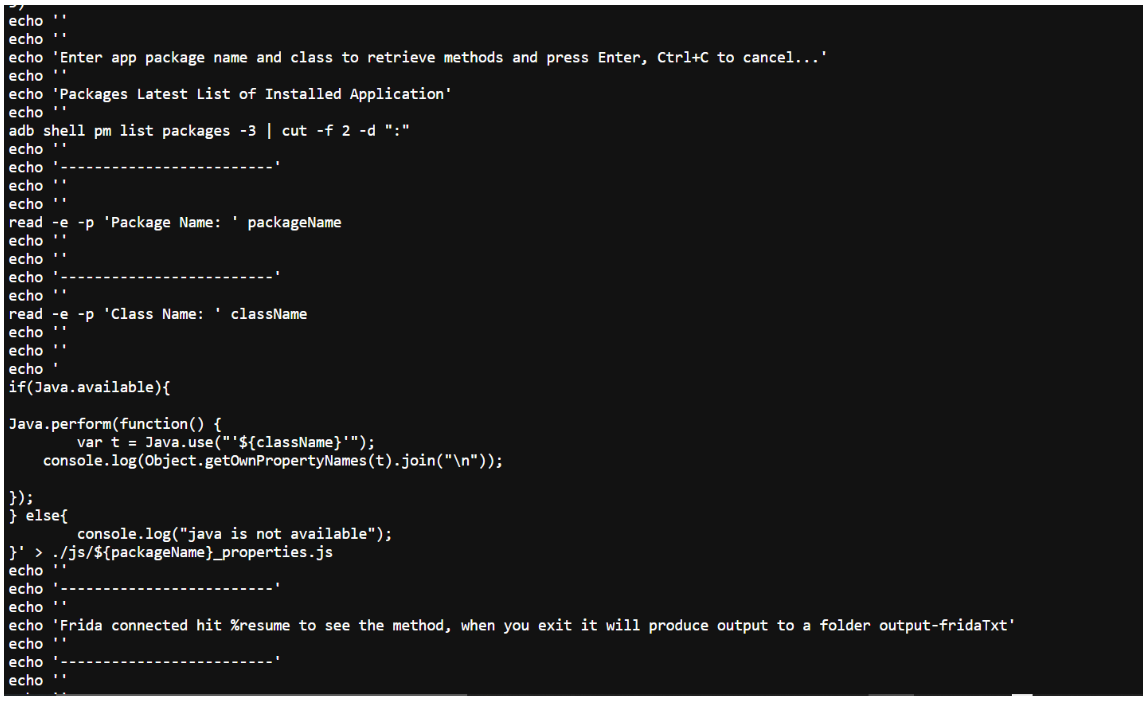
```
echo ''
read -n 1 -s -r -p 'Select operation type, ESC to cancel...' key
echo ''
if [ "$key" == $'\e' ]; then
echo ''
echo ''
echo 'Operation canceled!'
echo ''
echo ''
exit 1
fi
clear
case $key in
1)
echo ''
echo ''
echo 'Enter app package name to retrieve classes and press Enter, Ctrl+C to cancel...'
echo ''
echo 'Packages Latest List of Installed Application'
echo ''
adb shell pm list packages -3 | cut -f 2 -d ":"
echo ''
read -e -p 'Package Name: ' packageName
echo ''
echo ''
echo ''
echo '
if(Java.available){

Java.perform(function() {

        Java.enumerateLoadedClasses({


                onMatch : function(className){
                        console.log(className);
                },
```



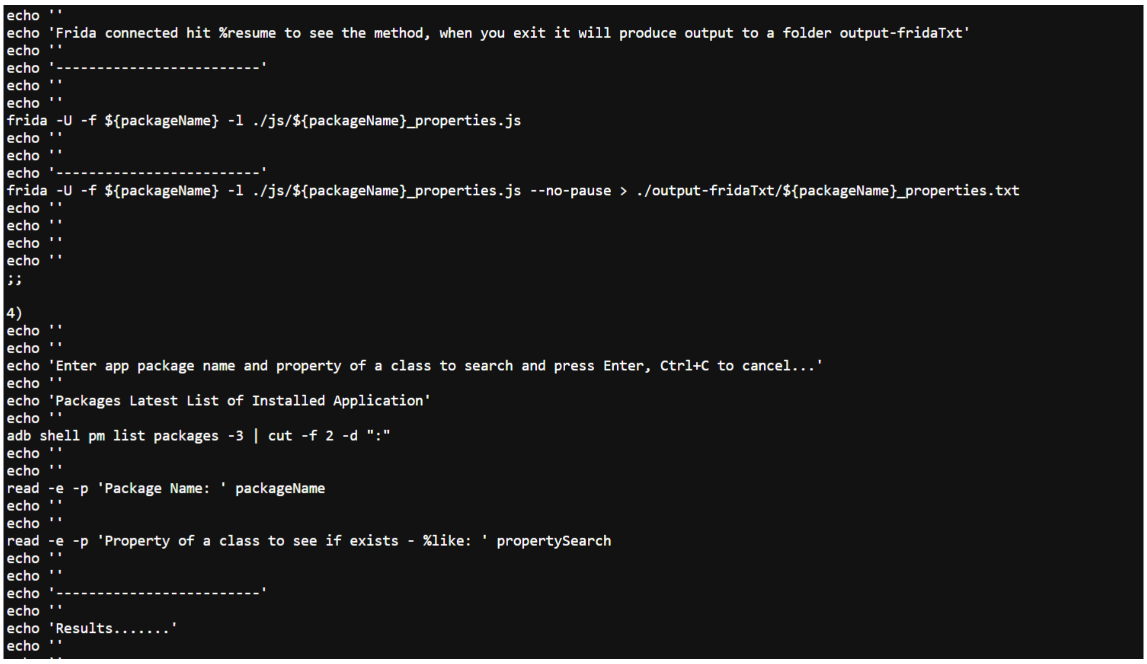
```
echo '
if(Java.available){

Java.perform(function() {

        Java.enumerateLoadedClasses({


                onMatch : function(className){
                        console.log(className);
                },
                onComplete : function(){
                        console.log("thank you");
                }
        })
});

} else{

        console.log("java is not available");
}' > ./js/${packageName}_class.js

echo '------------------------'
echo ''
echo 'Frida connected hit %resume to see the classes, when you exit it will produce output to a folder output-fridaTxt'
echo ''
echo '------------------------'
echo ''
echo ''
frida -U -f ${packageName} -l ./js/${packageName}_class.js
echo ''
echo ''
echo '------------------------'
echo ''
echo ''
echo ''
echo ''
frida -U -f ${packageName} -l ./js/${packageName}_class.js --no-pause > ./output-fridaTxt/${packageName}_class.txt
```



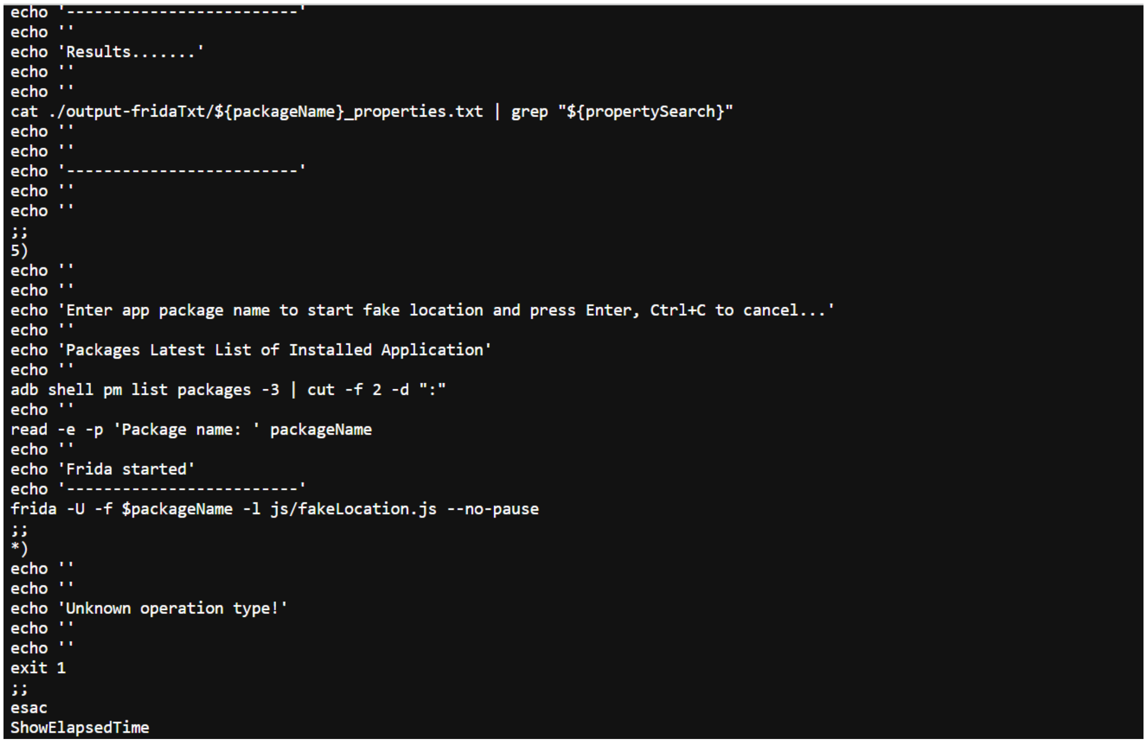
```
echo
frida -U -f ${packageName} -l ./js/${packageName}_class.js --no-pause > ./output-fridaTxt/${packageName}_class.txt
echo ''
echo ''
echo ''
echo ''
;;

2)
echo ''
echo ''
echo 'Enter app package name and class to search and press Enter, Ctrl+C to cancel...'
echo ''
echo 'Packages Latest List of Installed Application'
echo ''
adb shell pm list packages -3 | cut -f 2 -d ":"
echo ''
echo ''
read -e -p 'Package Name: ' packageName
echo ''
echo ''
read -e -p 'Class to see if exists - %like: ' classSearch
echo ''
echo ''
echo '------------------------'
echo ''
echo 'Results.......'
echo ''
echo ''
cat ./output-fridaTxt/${packageName}_class.txt | grep "${classSearch}"
echo ''
echo ''
echo '------------------------'
echo ''
echo ''
;;

3)
```



```
echo ''
echo ''
echo 'Enter app package name and class to retrieve methods and press Enter, Ctrl+C to cancel...'
echo ''
echo 'Packages Latest List of Installed Application'
echo ''
adb shell pm list packages -3 | cut -f 2 -d ":"
echo ''
echo '------------------------'
echo ''
echo ''
read -e -p 'Package Name: ' packageName
echo ''
echo ''
echo '------------------------'
echo ''
read -e -p 'Class Name: ' className
echo ''
echo ''
echo '
if(Java.available){

Java.perform(function() {
        var t = Java.use("'${className}'");
    console.log(Object.getOwnPropertyNames(t).join("\n"));

});
} else{
        console.log("java is not available");
}' > ./js/${packageName}_properties.js
echo ''
echo '------------------------'
echo ''
echo 'Frida connected hit %resume to see the method, when you exit it will produce output to a folder output-fridaTxt'
echo ''
echo '------------------------'
echo ''
```



```
echo ''
echo 'Frida connected hit %resume to see the method, when you exit it will produce output to a folder output-fridaTxt'
echo ''
echo '------------------------'
echo ''
echo ''
frida -U -f ${packageName} -l ./js/${packageName}_properties.js
echo ''
echo ''
echo '------------------------'
frida -U -f ${packageName} -l ./js/${packageName}_properties.js --no-pause > ./output-fridaTxt/${packageName}_properties.txt
echo ''
echo ''
echo ''
echo ''
;;

4)
echo ''
echo ''
echo 'Enter app package name and property of a class to search and press Enter, Ctrl+C to cancel...'
echo ''
echo 'Packages Latest List of Installed Application'
echo ''
adb shell pm list packages -3 | cut -f 2 -d ":"
echo ''
echo ''
read -e -p 'Package Name: ' packageName
echo ''
echo ''
read -e -p 'Property of a class to see if exists - %like: ' propertySearch
echo ''
echo ''
echo '------------------------'
echo ''
echo 'Results.......'
echo ''
```



```
echo '------------------------'
echo ''
echo 'Results.......'
echo ''
echo ''
cat ./output-fridaTxt/${packageName}_properties.txt | grep "${propertySearch}"
echo ''
echo ''
echo '------------------------'
echo ''
echo ''
;;
5)
echo ''
echo ''
echo 'Enter app package name to start fake location and press Enter, Ctrl+C to cancel...'
echo ''
echo 'Packages Latest List of Installed Application'
echo ''
adb shell pm list packages -3 | cut -f 2 -d ":"
echo ''
read -e -p 'Package name: ' packageName
echo ''
echo 'Frida started'
echo '------------------------'
frida -U -f $packageName -l js/fakeLocation.js --no-pause
;;
*)
echo ''
echo ''
echo 'Unknown operation type!'
echo ''
echo ''
exit 1
;;
esac
ShowElapsedTime
```

This script is a bash script that automates Frida commands. Frida is a dynamic instrumentation toolkit for developers, reverse engineers, and security researchers. This script is designed to help users quickly and easily run Frida commands on their device, without having to manually enter the commands.

The script starts by clearing the terminal and recording the current time. It then defines a function called "ShowElapsedTime", which calculates and displays the elapsed time since the script started running.

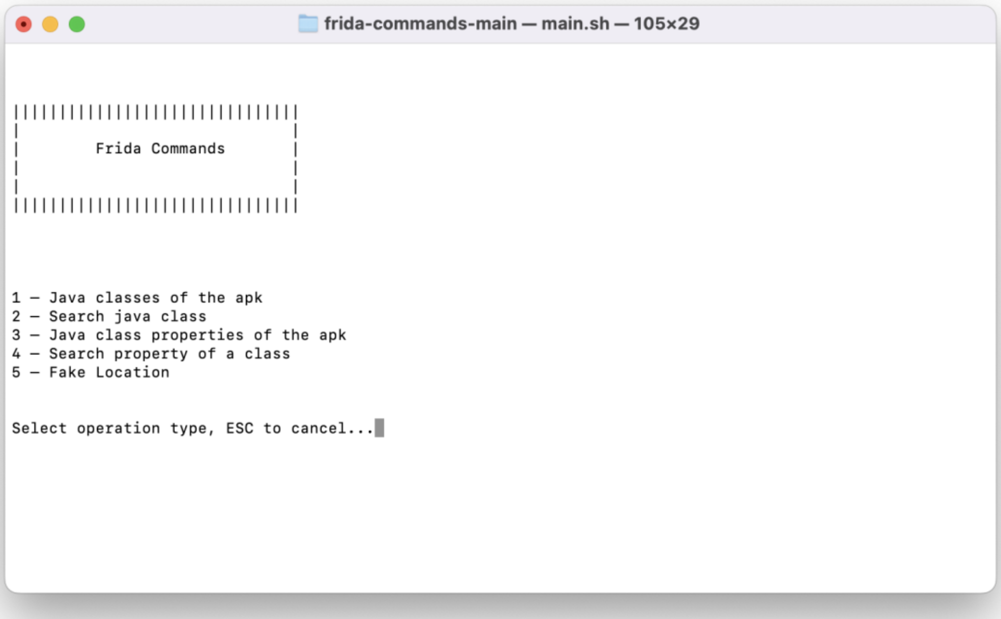
The script then displays a menu of options for the user to choose from. These options include retrieving the Java classes of an APK, searching for a specific Java class, retrieving the properties of a Java class, searching for a property of a class, and faking a location. The user can select an option by entering the corresponding number, or they can cancel the operation by pressing the ESC key.
If the user selects option 1, the script prompts the user to enter the package name of the app they want to retrieve classes for. The script then generates a JavaScript file that uses Frida to enumerate the loaded classes of the app and prints them to the console. The script then runs the Frida command to connect to the device and run the JavaScript file, and it also saves the output to a text file in a folder called "output-fridaTxt".
If the user selects option 2, the script prompts the user to enter the package name of the app and a class name to search for. It then searches the output text file from option 1 for the specified class name and displays the results.
Option 3 is similar to option 2, but it retrieves the methods of a class. Option 4 and 5 are also similar, but they search for a property and fake a location respectively.
Overall, this script provides a convenient way for users to automate Frida commands and easily retrieve information about an app's classes, properties and methods and also fake location. It also helps to reduce the time required to run Frida commands and make it easy for users to access the information they need.

You should follow the following steps to use it :

1. Start frida server
2. Run the command: chmod +x main.sh
3. Finally run: ./main

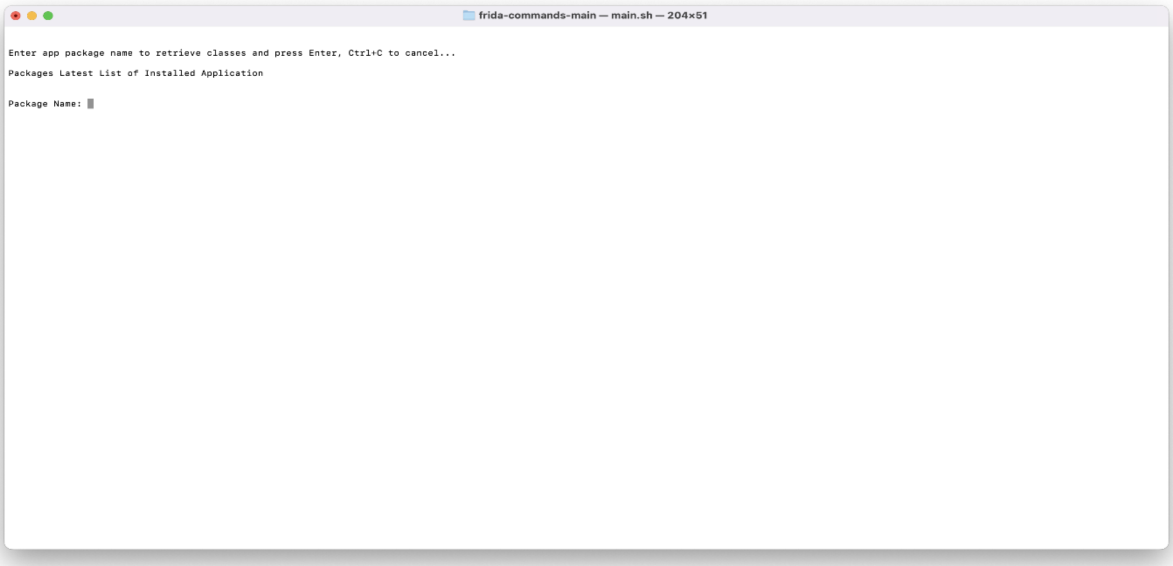
c. The application is currently being used

The result will be



```
frida-commands-main — main.sh — 105×29

|||||||||||||||||||||||||||||||
|                             |
|       Frida Commands        |
|                             |
|                             |
|||||||||||||||||||||||||||||||



1 – Java classes of the apk
2 – Search java class
3 – Java class properties of the apk
4 – Search property of a class
5 – Fake Location


Select operation type, ESC to cancel...
```

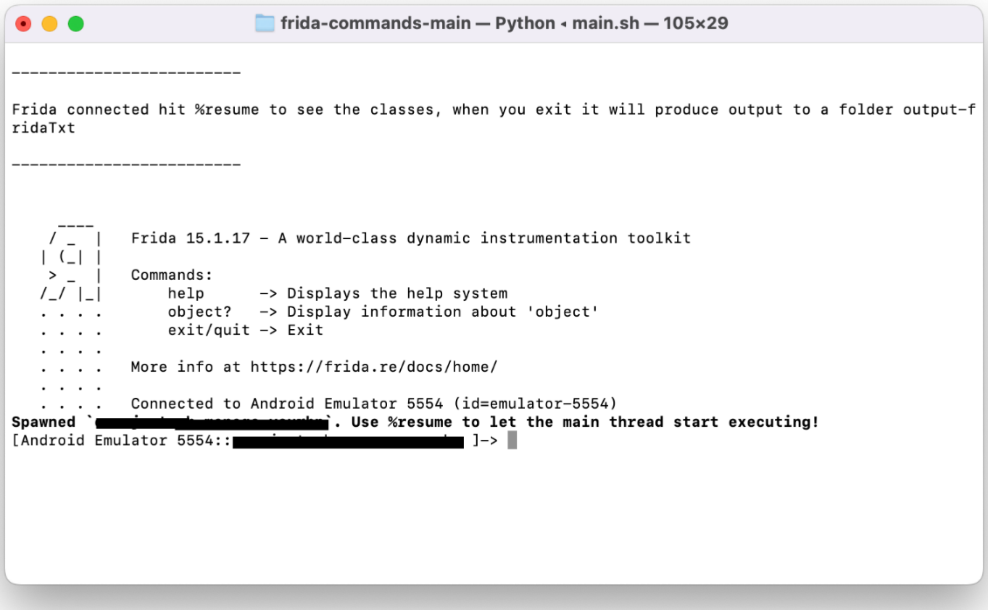
It will show you a list of all the installed packages from the emulator, and you should choose a package from the list in order to see all the classes that are available.



```
frida-commands-main — main.sh — 204×51

Enter app package name to retrieve classes and press Enter, Ctrl+C to cancel...
Packages Latest List of Installed Application

Package Name:
```

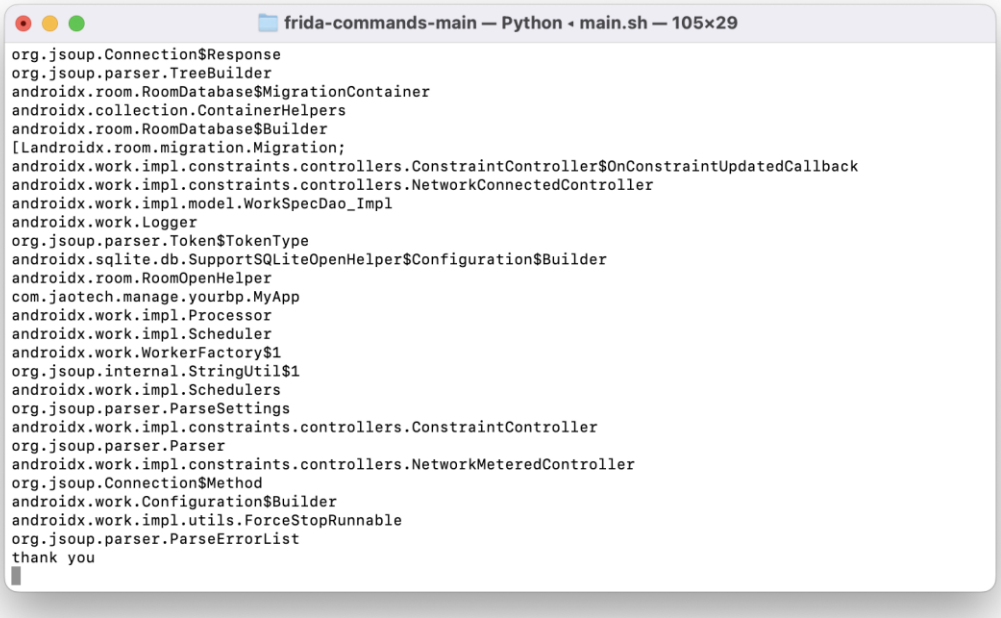
As you are in the frida terminal, you need to enter the command %resume in order to see all the classes, and after you enter the command exit, all the classes will be saved inside the output-fridaTxt folder.

The name of the package, as well as the name of the class that you are looking for, should be typed into the search field when you click number 2 in the menu to search for the class. Once you have completed this step, click enter in order to view the results of your search. You can download the tool from github following this link https://github.com/mathias82/frida-commands



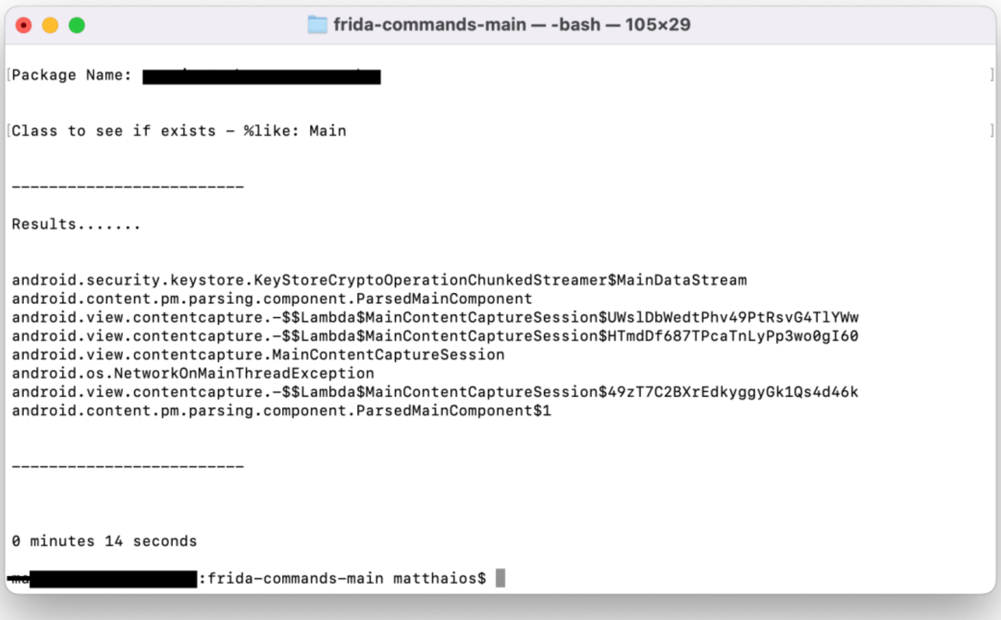
frida-commands-main — Python ◂ main.sh — 105×29

```
-------------------------

Frida connected hit %resume to see the classes, when you exit it will produce output to a folder output-f
ridaTxt

-------------------------



     ____
    / _  |   Frida 15.1.17 - 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)
Spawned `[redacted]`. Use %resume to let the main thread start executing!
[Android Emulator 5554::[redacted] ]->
```



frida-commands-main — Python ◂ main.sh — 105×29

```
org.jsoup.Connection$Response
org.jsoup.parser.TreeBuilder
androidx.room.RoomDatabase$MigrationContainer
androidx.collection.ContainerHelpers
androidx.room.RoomDatabase$Builder
[Landroidx.room.migration.Migration;
androidx.work.impl.constraints.controllers.ConstraintController$OnConstraintUpdatedCallback
androidx.work.impl.constraints.controllers.NetworkConnectedController
androidx.work.impl.model.WorkSpecDao_Impl
androidx.work.Logger
org.jsoup.parser.Token$TokenType
androidx.sqlite.db.SupportSQLiteOpenHelper$Configuration$Builder
androidx.room.RoomOpenHelper
com.jaotech.manage.yourbp.MyApp
androidx.work.impl.Processor
androidx.work.impl.Scheduler
androidx.work.WorkerFactory$1
org.jsoup.internal.StringUtil$1
androidx.work.impl.Schedulers
org.jsoup.parser.ParseSettings
androidx.work.impl.constraints.controllers.ConstraintController
org.jsoup.parser.Parser
androidx.work.impl.constraints.controllers.NetworkMeteredController
org.jsoup.Connection$Method
androidx.work.Configuration$Builder
androidx.work.impl.utils.ForceStopRunnable
org.jsoup.parser.ParseErrorList
thank you
```



frida-commands-main — -bash — 105×29

```
Package Name: [redacted]

Class to see if exists - %like: Main

-------------------------

Results.......

android.security.keystore.KeyStoreCryptoOperationChunkedStreamer$MainDataStream
android.content.pm.parsing.component.ParsedMainComponent
android.view.contentcapture.-$$Lambda$MainContentCaptureSession$UWslDbWedtPhv49PtRsvG4TlYWw
android.view.contentcapture.-$$Lambda$MainContentCaptureSession$HTmdDf687TPcaTnLyPp3wo0gI60
android.view.contentcapture.MainContentCaptureSession
android.os.NetworkOnMainThreadException
android.view.contentcapture.-$$Lambda$MainContentCaptureSession$49zT7C2BXrEdkyggyGk1Qs4d46k
android.content.pm.parsing.component.ParsedMainComponent$1

-------------------------

0 minutes 14 seconds

[redacted]:frida-commands-main matthaios$
```

6. Conclusion

In conclusion, using Frida and visual representation techniques can be an effective method for detecting Android malware network traffic. Frida allows analysts to dynamically instrument and monitor the behavior of Android apps, providing valuable insights into their inner workings. By combining this with visual representation tools such as flow charts, heat maps, and graphs, analysts can more easily identify patterns and anomalies in network traffic that may indicate the presence of malware. This approach can provide a more intuitive and easy-to-understand way for non-technical stakeholders to comprehend the results of the analysis. Overall, using Frida and visual representation can be a powerful combination in the fight against Android malware.

7. References

1. *Application signing : Android Open Source Project*. Android Open Source Project. (n.d.). Retrieved December 4, 2022, from https://source.android.com/security/apksigning

2. *Application sandbox : Android Open Source Project*. Android Open Source Project. (n.d.). Retrieved December 4, 2022, from https://source.android.com/docs/security/app-sandbox

3. *Security-enhanced linux in Android : Android Open Source Project*. Android Open Source Project. (n.d.). Retrieved December 4, 2022, from https://source.android.com/docs/security/features/selinux

4. *What is phishing: Attack Techniques & Scam examples: Imperva* (2020) *Learning Center*. Available at: https://www.imperva.com/learn/application-security/phishing-attack-scam/#:~:text=Phishing%20is%20a%20type%20of,instant%20mes-sage%2C%20or%20text%20message. (Accessed: January 17, 2023).

5. Google. (n.d.). *Malware categories | play protect | google developers*. Google. Retrieved December 4, 2022, from https://developers.google.com/android/play-protect/phacategories

6. Google. (n.d.). *Mobile unwanted software (muws) | play protect | google developers*. Google. Retrieved December 4, 2022, from https://developers.google.com/android/play-protect/mobile-unwanted-software

7. j.R. Vacca. Computer and information security handbook. Morgan Kaufmann, 2017.

8. Bertolis. (2022, March 21). *A step-by-step Android Penetration Testing Guide for Beginners*. Hack The Box. Retrieved December 5, 2022, from https://www.hackthebox.com/blog/intro-to-mobile-pentesting

9. Fri, P. (2021, June 24). *Pentesting Android apps using Frida*. NotSoSecure. Retrieved December 21, 2022, from https://notsosecure.com/pentesting-android-apps-using-frida

10. *Understanding app permissionsedit pagepage history* (no date) *Understanding App Permissions CodePath Android Cliffnotes*. Available at: https://guides.codepath.com/android/Understanding-App-Permissions (Accessed: January 17, 2023).

11. Arctic Wolf (2022) *Most common malware*, *Arctic Wolf*. Available at: https://arcticwolf.com/resources/blog/8-types-of-malware/ (Accessed: January 17, 2023).

12. Google - Android Security & Privacy Year in Review 2020: https://android-developers.googleblog.com/2021/01/android-security-privacy-year-in-review-2020.html.

13. Resource center (no date) Kaspersky. Available at: https://www.kaspersky.com/resource-center/threats/billing-fraud-on-android (Accessed: January 17, 2023).

14. Kaspersky (2022) *What is smishing and how to defend against it*, *www.kaspersky.com*. Available at: https://www.kaspersky.com/resource-center/threats/what-is-smishing-and-how-to-defend-against-it (Accessed: January 17, 2023).

15. Crane, C. (2021) *What is vishing? how to recognize voice phishing phone calls in 2021*, *Hashed Out by The SSL Store™*. Available at: https://www.thesslstore.com/blog/what-is-vishing-how-to-recognize-voice-phishing-phone-calls/ (Accessed: January 17, 2023).

16. *Resource center* (no date) *Kaspersky*. Available at: https://www.kaspersky.com/resource-center/threats/ransomware-on-android (Accessed: January 17, 2023).