

UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Karl Erik Mander

Intercepting Mobile-ID SIM Toolkit Calls On Android

Bachelor's Thesis (9 ECTS)

Supervisor: Arnis Paršovs, PhD

Tartu 2023

Intercepting Mobile-ID SIM Toolkit Calls On Android

Abstract:

This thesis investigates the security risk of intercepting Mobile-ID SIM Toolkit calls on Android. The investigation is done by modifying the Android operating system with malware. Through an in-depth analysis of the communication protocol between an Android phone and a SIM card, this study demonstrates that attackers who have gained access to the victim's phone through illegitimate apps or other exploits with root privileges may be able to remotely control Mobile-ID operations by intercepting SIM card communications. From there on, the system could complete all Mobile-ID transactions surreptitiously and automatically. This thesis aimed to research the security architecture of Android OS concerning Mobile-ID and discuss possible options that a malware creator would have to implement to achieve SIM command intercepting capabilities.

Keywords:

Mobiil-ID, SMS, SIM-kaart, Malware, Android

CERCS:

P170 Computer science, numerical analysis, systems, control

Mobiil-ID SIM-i tööriistakomplekti käskude pealtkuulamine Androidis

Lühikokkuvõte:

Antud lõputöö uurib Androidi Mobiil-ID SIM-i tööriistakomplekti käskude pealtkuulamise turvariske. Uurimine toimub Androidi operatsioonisüsteemi muutmise teel pahavaraga. Süvitsi analüüsid Androidi telefoni ja SIM-kaardi vahelist suhtlusprotokolli, näitab see uuring, et ründajad, kes on saanud juurdepääsu ohvri telefonile läbi ebaseaduslike rakenduste või muude meetodite, võivad kaugjuhtimisega kontrollida Mobile-ID toiminguid. Sealt edasi saab süsteem kõik Mobiil-ID tehingud varjatult ja automaatselt sooritada. Käesolev lõputöö eesmärk oli uurida Androidi operatsioonisüsteemi turvaarhitektuuri seoses Mobile-ID-ga ning arutada võimalikke võimalusi, mida pahavara looja peaks SIM-käskude pealtkuulamise võimaluste saavutamiseks rakendama.

Võtmesõnad:

Mobiil-ID, SMS, SIM-kaart, Android, Pahavara

CERCS:

P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimise teooria)

Table of Contents

| | |
|---|-----------|
| 1. Introduction..... | 5 |
| 2. Mobile-ID malware..... | 6 |
| 2.1. Mobile-ID architecture..... | 6 |
| 2.2. Mobile-ID STK call interception malware | 9 |
| 3. Development of the proof-of-concept Mobile-ID malware | 14 |
| 3.1 Hardware and software setup..... | 14 |
| 3.1.1 Android device and rooting of it..... | 14 |
| 3.1.2 Android Debug Bridge (ADB)..... | 16 |
| 3.1.3 SSH | 16 |
| 3.1.4 Dalvik Virtual Machine | 18 |
| 3.1.5 Running Java on Android | 19 |
| 3.1.6 Installation script..... | 20 |
| 3.2 Memory dump..... | 21 |
| 3.3 Patching Stk.apk | 22 |
| 3.4 JAttach | 25 |
| 3.5 Android Interface Definition Language (AIDL)..... | 26 |
| 3.6 Radio Interface Layer Daemon (RILD)..... | 26 |
| 3.7 Binder..... | 28 |
| 4. Discussion..... | 29 |
| 5. Conclusion | 31 |
| References | 32 |
| Appendices..... | 39 |
| A Code and log snippets | 39 |
| B Licence | 43 |

1. Introduction

The rise of digital technology has transformed how people conduct business and interact with the outside world. Thus, increasing reliance on digital communication and transactions has increased the demand for secure and dependable online authentication methods. In response, the Estonian government developed eID, a virtual electronic identity that facilitates secure digital authentication and signature [1]. Mobile-ID, which Estonians extensively use, is a platform that provides eID services, such as authentication and the ability to create digital signatures [2]. However, the increasing prevalence of OS-level malware that can exploit operating system vulnerabilities threatens the security of Mobile-ID transactions [3]. The Mobile-ID trojan would be a hazard because it could eavesdrop on Mobile-ID transactions and manipulate them to obtain illicit access to user accounts. This thesis investigates the viability of Android-based malware that can exploit the vulnerability of Mobile-ID communicating SIM Toolkit commands in plain text. The goal is to construct an Android proof-of-concept (POC) application that runs inside the Android operating system that can hijack a user's Mobile-ID session from plain text SIM Toolkit commands and grant illicit access to the user's account. Due to Android's availability and wide usage, the malware is designed to run on Android [4]. The thesis is structured to provide a comprehensive understanding of the topics related to investigating Mobile-ID malware. Mobile-ID and the Android operating system are introduced in the second chapter with a discussion of the threat posed by Mobile-ID-targeting OS-level malware. The chapter's first section concentrates on the architecture of Mobile-ID. The second section introduces possible attack vectors that could be used to intercept Mobile-ID transactions. The third chapter describes the creation attempts of POC malware, as well as the obstacles and escapades encountered during its implementation. The investigation is concluded by analysing the findings and drawing conclusions.

2. Mobile-ID malware

This chapter explains how Mobile-ID works on the user level, how it employs Android STK for transaction confirmation and PIN entry and describes the theoretical system architecture of Mobile-ID. The chapter introduces malware that can intercept and hijack Mobile-ID, discusses its impact, required permissions, and deployment methods, and contrasts its methods with relevant work.

2.1. Mobile-ID architecture

The Estonian state offers an eID solution for digital authentication, i.e., a virtual electronic identity document that allows all people or services to authenticate securely via the Internet [5]. Logging in with an eID is equivalent to showing a document in a store or government institution in the virtual world. EID allows virtual user identification and signing equivalent to a real-world signature using two secret cryptographic signing keys [6]. The first solution was introduced in 2002 as an ID card and is still in use [7]. Mobile-ID is one of the three technical solutions offering eID, which allows Estonians to enter eID-supporting services and digitally sign documents quickly. Mobile-ID includes a special SIM card with which it is possible to enter services or sign documents from any device [8]. The entire authentication process begins with mobile service providers in Estonia. Customers must initially enrol in a Mobile-ID subscription to receive a special SIM card (see Figure 1). The SIM card contains two private keys, one for authentication and the second for digital signature creation.



Figure 1. Telia's Mobile-ID SIM card with PINs, PUK, and handwritten phone number

Second, the SIM card must be inserted into a phone that complies with international STK GSM standards [9]. SIM Application Toolkit (STK) is a standard that allows for the exchange of a variety of instructions between a phone and a SIM card. Using Mobile-ID, the user can authenticate in any application that supports it by identifying with their Mobile-ID phone number. This action triggers the verification prompt containing the control code on both the web service and the phone (see Figures 2 and 3). After the user confirms that the control codes are identical, the user is prompted to input the associated PIN (see Figure 4). PIN 1 is used for authentication, while PIN 2 is used for document signature. SIM card then verifies the validity of the PIN. The SIM then signs the data received from the web service and sends the signed response to the mobile service, which is forwarded to the initial web service. Afterward, the web service validates the signature and provides the user access to their account on the web service.

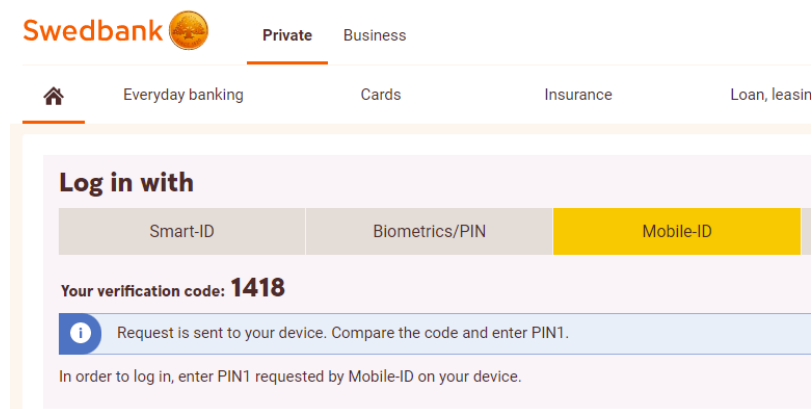


Figure 2. User verification on the application with the help of control code

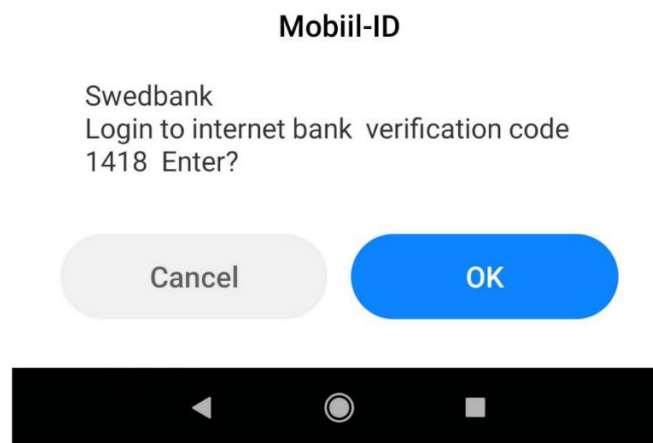


Figure 3. User verification on Android device with the help of a control code

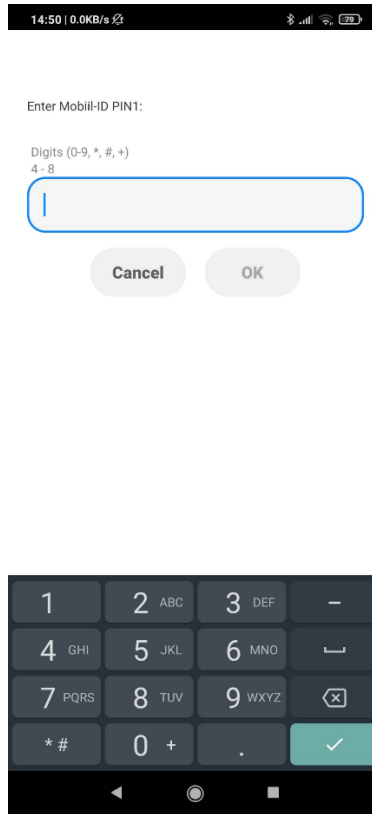


Figure 4. The entering of Mobile-ID PIN1

The ability in Mobile-ID technical platform to request Mobile-ID PINs from the user's phone is enabled by STK commands. When the user begins the authentication login process, the mobile operator sends a Mobile-ID service SMS to the user's mobile phone. The phone forwards the received service SMS to the SIM card. SIM then transmits an STK command to the phone to display the affirmation dialog, followed by the PIN input dialog. After the user has entered the appropriate PIN, the SIM card then verifies the PIN and cryptographically signs the mobile operator's request using the appropriate signature keys. The resulting signature is forwarded back to the mobile provider, which forwards it to the web service that started the authentication process [10].

For STK commands to function on a smartphone, some service must listen for those commands. For Android, `StkAppService` in `Stk.apk` is responsible for receiving and handling STK commands. The STK app is pre-installed on Android devices as a system app. It enables users to communicate with the SIM card via an interface that displays forms created by STK messages. Mobile-ID uses STK commands to communicate with the user through mobile phone dialog

prompts. For example, `DISPLAY_TEXT` is used for displaying text on the phone. `Stk.apk` is a standalone application on Android, but the overall architecture (see Figure 5) is that the SIM chip transmits an application protocol data unit (APDU) message to Androids `TelephonyService`, and `TelephonyService` handles it. When the `TelephonyService` detects an STK command, it is sent using a binder to the `StkAppService`. After the `StkAppService` has completed the actions associated with the STK command (showing a text message or asking for text input on screen) and received the result from the phone, it sends a direct response instance to the `TelephonyService`. The command is complete when `TelephonyService` dispatches the STK result command back to SIM.

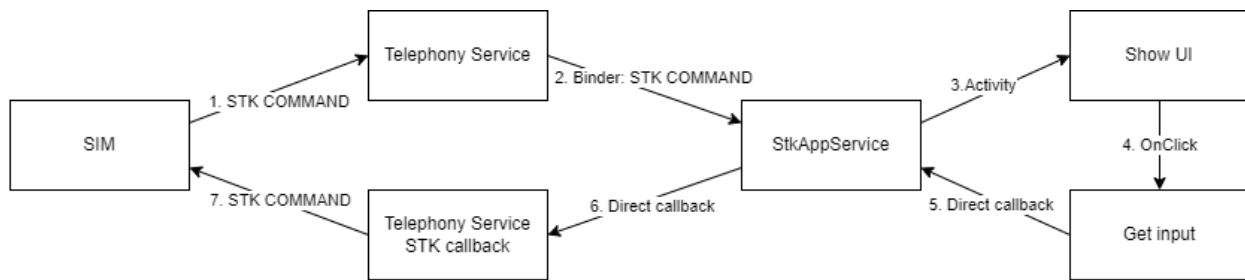


Figure 5. Handling of STK commands

The conclusion is that Mobile-ID extensively relies on STK commands for user interaction that malicious actors can eavesdrop on and modify.

2.2. Mobile-ID STK call interception malware

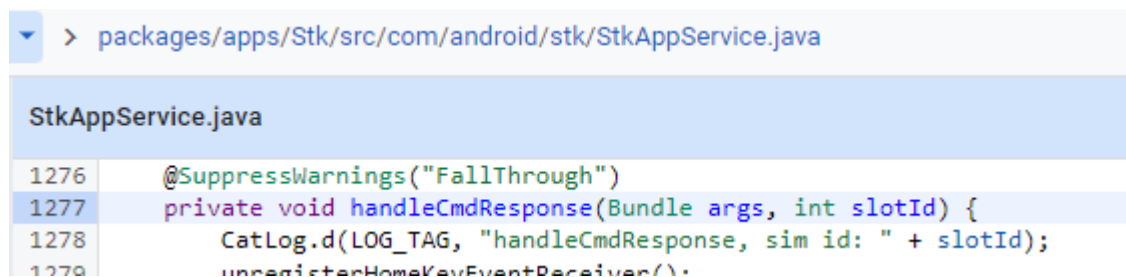
The goals for Mobile-ID STK command intercepting malware are to embed itself into the user's smartphone, learn the user's Mobile-ID PINs by sniffing Mobile-ID STK commands, and have the functionality to maliciously complete the bad actor's Mobile-ID authentication requests by responding to Mobile-ID STK commands without alerting the user about ongoing authentication. Options of how to remotely control the malware are discussed later in this chapter. The first step for the malware would be to learn the user's Mobile-ID PINs during a legitimate transaction where the user enters these PINs. Then those learned PINs can be maliciously used to complete bad actor-initiated Mobile-ID transactions automatically.

After careful consideration, the most suitable place to intercept STK commands is the `Stk.apk StkAppService` class [11]. The Telephony service could also be utilized to intercept STK commands as it would produce identical results to intercepting `StkAppService`. However, the Telephony service was not selected because it has a significantly larger code footprint and processes other data like APDU traffic [12]. The `StkAppService` point of interest methods are `handleCmd()` (see Figure 6) [13] and `handleCmdResponse()` (see Figure 7) [14].



```
packages/apps/Stk/src/com/android/stk/StkAppService.java
StkAppService.java
1064
1065     private void handleCmd(CatCmdMessage cmdMsg, int slotId) {
1066
```

Figure 6. The `handleCmd` method signature in `StkAppService.java` class hosted on Android Code Search



```
packages/apps/Stk/src/com/android/stk/StkAppService.java
StkAppService.java
1276     @SuppressWarnings("FallThrough")
1277     private void handleCmdResponse(Bundle args, int slotId) {
1278         CatLog.d(LOG_TAG, "handleCmdResponse, sim id: " + slotId);
1279         unregisterHomeKeyEventReceiver();
```

Figure 7. The `handleCmdResponse` method signature in `StkAppService.java` class hosted on Android Code Search

As it processes all incoming STK commands, the `handleCmd()` method is the principal entry point for the business logic of `StkAppService`. The primary entry point for receiving response data from launched activities is the `handleCmdResponse()` method. `HandleCmdResponse()` is required because the activities launched by `handleCmd()` are asynchronous. Because asynchronous commands are unpredictable in terms of duration, the `handleCmd()` method transfers control of the command response to `handleCmdResponse()`. Thus, there are no stalling components, and the service can operate efficiently. In the showing of the confirmation dialog (see Figure 3), the `DISPLAY_TEXT` STK

command is used. `handleCmd()` creates an activity and shows it. The same applies to the `GET_INPUT` STK command [10]. By overriding the `handleCmdResponse()`, detecting, sniffing, and learning the user-entered PIN 1 and 2 (see Figure 8) is possible. Learning the PINs is crucial for creating malware that can maliciously respond to incoming transactions.

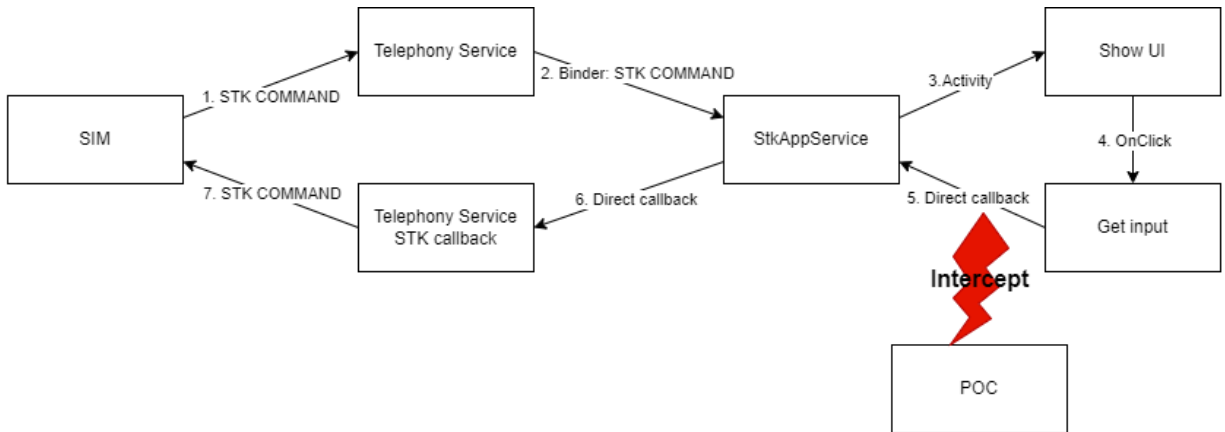


Figure 8. Intercepting the input activity's result

Having control over the `handleCmd()` method is the primary point in creating the automatic Mobile-ID STK command response system. The `handleCmd()` method comes with arguments called `cmdMsg` with the type `CatCmdMessage` and `slotId` with the type `int`. The critical information is held inside the `cmdMsg` variable. It holds the command type, dialog text, input activity text, and the maximum length of input digits. This information makes filtering out all the STK messages possible so that the malware would only react to `DISPLAY_TEXT` and `GET_INPUT` commands. Additional filtering can be added for detecting the string “Mobile-ID” in the title (see Figure 3). The malware could automatically respond to `DISPLAY_TEXT` when both filter conditions are met by instantly responding to the Telephony service with the `RES_ID_CONFIRM` command. The same logic can be applied for the `GET_INPUT` command so that the malware would respond with the Mobile-ID PIN and the confirmation (see Figure 4). The Mobile-ID pin entry dialog title can be used to detect if the Mobile-ID is requesting PIN 1 or 2, as it contains the requested PIN type (see Figures 4 and 9).

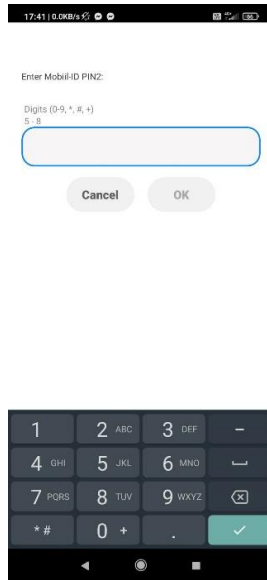


Figure 9. The entering of Mobile-ID PIN2

A command and control (C&C) server could be used for controlling the malware. This would give more comprehensive statistics of the malware, like how much the user uses Mobile-ID for which services through the Mobile-ID confirmation prompt (see Figure 3), whether the malware has learned the PINs, retrieving the user's phone number, and controlling the automatic transaction completion. C&C communication is ideal but would leave forensic material in the form of network communication for detection. SMS-based communication could replace internet communication so that if the malware learns the PINs, it sends an SMS to a predefined number. Afterward, a bad actor can send a special SMS that enables automatic Mobile-ID authentication confirmation. There could also be a night time window where all Mobile-ID authentications are automatically completed to not interfere with everyday usage and alert the user that something malicious is happening. This would eliminate the need to signal the malware to enable automatic authentication completion. However, due to the necessity of knowing the victim's Mobile-ID phone number for starting the Mobile-ID authentication, some communication must forward the victim's phone number to the bad actor. The forwarding leaves behind forensic material like the predefined phone number. The only method to automatically complete the authentication flow with significantly reduced forensic data would be to target users whose mobile phone numbers are known and set the automatic completion to occur at night to avoid interfering with daily usage. Attacking a predefined victim with a known phone number eliminates any chance of tracing the attack back to the bad actor.

In order to intercept STK commands, malware needs root privileges on the victim's phone. With root privileges, the malware can access the whole Android operating system and manipulate every aspect of processes. The malware needs root privileges for STK command intercepting features to work.

There are multiple ways to infect a smartphone with malware. The malware could be deployed through a remote code-executing attack like a 2020-reported vulnerability that enabled malicious code execution through Bluetooth-enabled devices [15]. There are numerous cases where malicious applications have passed Google's malware checks and have been hosted on Google Play [16]. "Juice Jacking" is another possibility for malware infiltration [17]. The risk is that by using public charging stations, the station can load malware through the charging cable into the smartphone. Smartphones should always be safeguarded, as physically compromised phones can be infected by booting the device into recovery mode or substituting a motherboard chip.

Smart-ID is a technological innovation that enables Estonian, Latvian and Lithuanian citizens to verify their identity using eID [18]. Smart-ID is distinct from Mobile-ID in that it employs an Android application to establish communication with its users. Nevertheless, the fundamental mechanism for verifying identity and signing documents in the core remains unchanged, relying on a dual private key approach. Silver Maala has developed a POC malware for Smart-ID that can acquire knowledge of PIN 1 and 2 by being installed in conjunction with Smart-ID. The malware can then subsequently insert the acquired PINs as required. The proposed Mobile-ID malware method significantly differs from the Smart-ID malware POC. The Smart-ID malware employs a user space application for executing the business logic, leverages OpenCV for acquiring knowledge of the Smart-ID PINs and utilizes manipulation of the smartphone touchscreen to simulate user input, as stated in reference [19]. Conversely, the hypothesized Mobile-ID malware has the potential to operate within the Android operating system and covertly engage with the Mobile-ID STK commands. The detection of this approach is considerably challenging owing to the limited access of the victim to the operating system space. In theory, the same approach can be used for Mobile-ID POC as used in Smart-ID POC, but this would mean that the Mobile-ID authentication dialogs will not be hidden from the victim.

3. Development of the proof-of-concept Mobile-ID malware

This chapter describes the software setup that was used for the development of the proof-of-concept (POC) malware and describes the hardware that was utilised to create it. Various substantial and minor approaches for the creation of POC malware are explained in detail.

3.1 Hardware and software setup

This section describes the methods used to create the POC Mobile-ID malware application. The necessary hardware and software are discussed.

3.1.1 Android device and rooting of it

“Rooting” an Android device means breaking into the system and gaining superuser privileges. By removing manufacturer and carrier restrictions, known as “rooting”, Android users can make extensive customizations to their devices [20]. It is necessary to have root privileges due to theorised Mobile-ID malware needing access to Android operating system services for manipulating STK commands.

The selection of the Target Android device was based on the device’s availability to the author of this work during the thesis writing period. There were three available phones (see Figure 10).

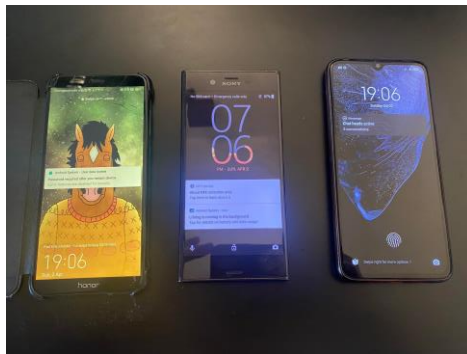


Figure 10. Phones used in the development. Honor at the left, Sony at the middle, and Xiaomi at the right

The initial one, an Honor 9 Lite LLD-L31, build number 9.1.0.168, Android version 9 with kernel version 4.9.148, was obtained to root it, but Huawei stopped giving out the required bootloader unlock codes so the device could not be rooted [21]. The next phone, SONY XPERIA F8331 running Android 8.0.0 with kernel version 3.18.66-perf-gd5ec1d77090d, was used to attempt to gain root shell access, but all CVE exploits failed or were for different kernel versions. Finally, the XIAOMI Mi 9 Lite running Android 10 QKQ1.190828.002 with kernel version 4.9.186-perf-G73C8056 was selected as the target device, as it had a solid rooting guide [22] available that worked (see Figure 11).

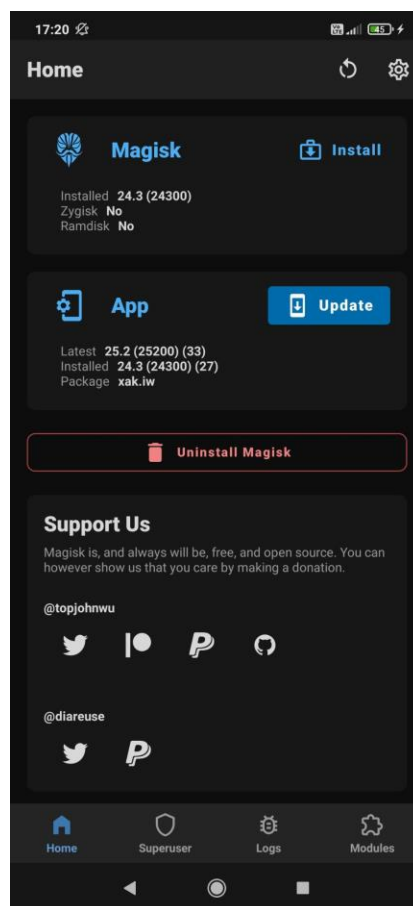


Figure 11. Screenshot of Magisk Root Manager status page indicating by the field “installed” that the device is rooted

After rooting the device, the smartphone functionality was evaluated to ensure nothing was damaged in the rooting progress. Titanium Backup was downloaded from the Google Play Store

to confirm that rooting was effective, as it requires root permissions for backing up installed apps and their data [23]. The backup was successful, indicating that root permissions are present.

3.1.2 Android Debug Bridge (ADB)

Android Debug Bridge, or ADB, is one of many Android Software Development Kit utilities that improve programmatic access to Android phones, both virtual and physical. When enabled from the phone, it can access user-level functionality and nearly complete system-level access. It is primarily employed by integrated development environments, or IDEs, to facilitate Android application development and monitoring on mobile devices [24]. It can also be used independently to generate installation scripts. An installation script has been developed to programmatically transfer and execute program files on Android devices using the `adb push` and `adb shell` commands. While SSH can execute commands and transfer files, the adb tools provide more robust functionality. Only connecting the Android smartphone to the host computer on which the script is executed eliminates the need for user-side configuration.

3.1.3 SSH

Secure Shell Protocol (SSH) was essential for establishing a secure and dependable terminal connection to the smartphone, as it enables secure remote access to the device's command line interface. SSH is an ideal choice for development and other technical duties due to its efficiency and simplicity. SSH was enabled on the device by installing the SSHHelper application, which allowed the setup of an SSH connection to the device from a remote terminal, and enabled the execution of commands and access to the device's file system [25]. SSH enabled root terminal access, granting the administrative rights required for additional investigation and testing (see Figure 12). Access to the root terminal was essential for further device analysis, including the examination of system files, execution of sophisticated commands, and customization of configurations (see Figure 13).

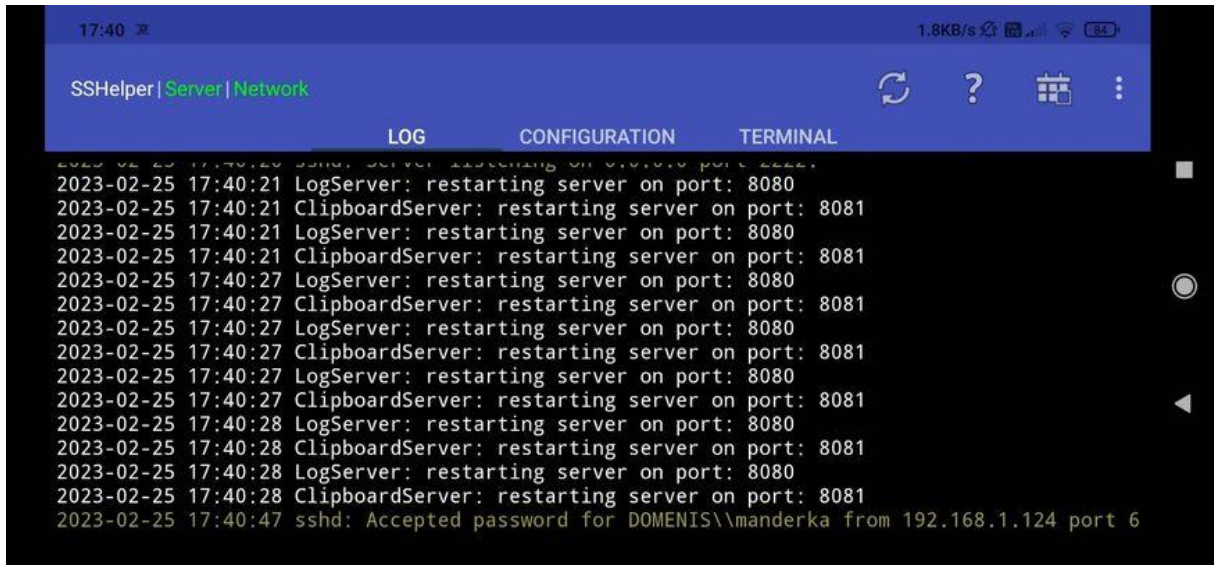


Figure 12. SSHHelper setup with a connected client

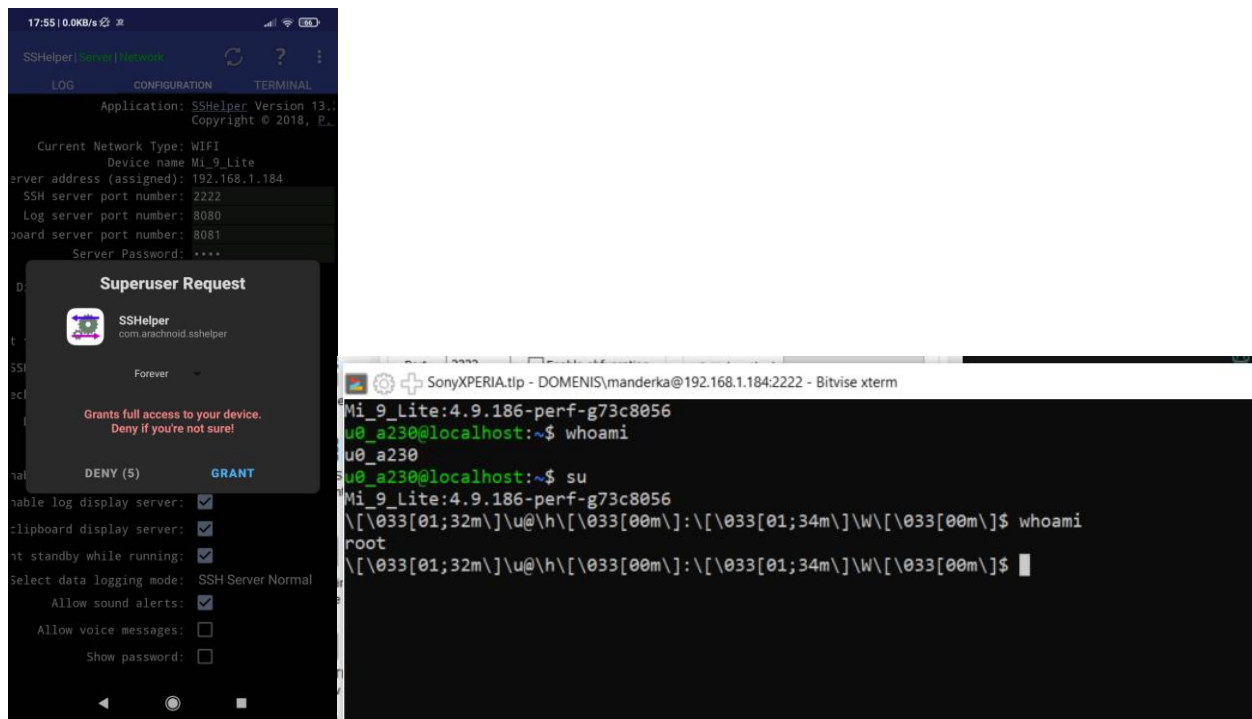


Figure 13. Requesting and getting root permission

Although Android Debug Bridge (ADB) could be used to connect to the Android terminal, the SSH terminal client Bitwise [26] provided a more user-friendly UI with an integrated SFTP browser for file browsing that was used extensively in the span of the development.

3.1.4 Dalvik Virtual Machine

Dalvik is a process virtual machine used to execute Android applications written in Java, and it was used for this thesis to create a POC application that could interact with existing Android operating system Java code through the means of Java’s integrated code manipulation methods. Dalvik VM was a necessary utility for Android versions up to 5.0 “Lollipop” before Android replaced it with Android Runtime (ART). Dalvik uses a register-based virtual machine more suitable for low-storage devices. ART takes a different approach to app execution by compiling Android programs to machine code, necessitating significantly more memory for enhanced performance [27]. Android application bytecode is stored in .dex files, and the Android dx or d8 utility converts Java.class files to .dex format (see Figure 14). The difference between dx and d8 is that d8 supports more recent Java 8 features, whereas dx is no longer supported [28].

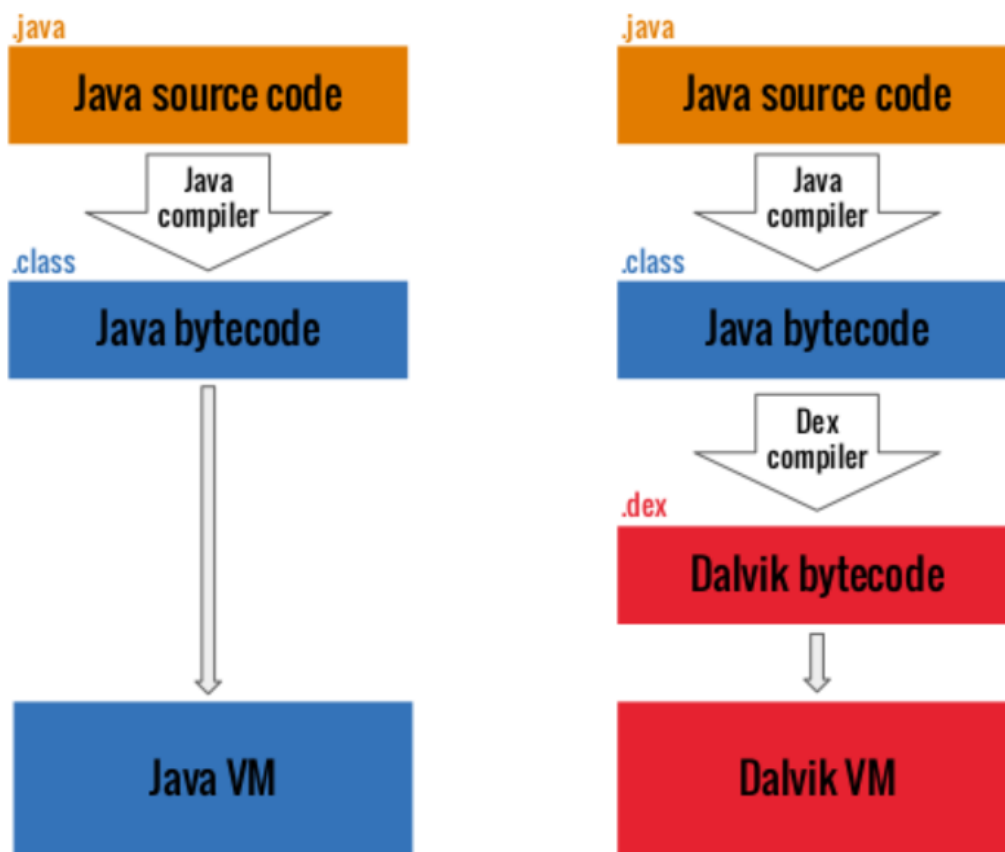


Figure 14. Comparison of running Java bytecode on JVM and DVM [29]

Sample d8 compilation command:

```
d8.bat --output=Program.jar
--lib "%path_to_sdk_folder%\platforms\android-33\android.jar"
--classpath "%path_to_java_classpath%\out\production\Program"
"%path_to_compiled_java_class%\out\production\Program\com\program\Program.class"
```

This command uses the d8 tool to convert a Java class file to an Android DEX byte-code format and package it as the “Program.jar” JAR file. “—output=Program.jar” specifies the name of the output JAR file. The “—lib” parameter specifies the Android SDK platform library’s location. The “—classpath” parameter specifies where the Java compiler should search for classes to compile. Lastly, the command specifies the location of the Java class file that will be converted to DEX byte-code format and packaged in the output JAR file [28].

3.1.5 Running Java on Android

Although it is mentioned in Section 3.1.4 that Dalvik VM was replaced by the ART after Android 5.0 “Lollipop” this thesis software setup uses Dalvik VM for running Java 8 code on Android. The aim for running Java on Android and not some other language like C is that it could interact with existing Android operating system Java code through the means of Javas integrated code manipulation methods. First, the Java code must be compiled into Java byte code using IDE or Java compiler like Javac. IntelliJ IDEA 2022.2.1 (Ultimate Edition) with JDK 8 and Android SDK 29 was used to compile the POC application. Compiled Java byte code should be converted to DEX format using the Android STK program d8 introduced in the 3.1.4 chapter. Now with the Dexed Java program, it can be run on Android using the following shell script [30]:

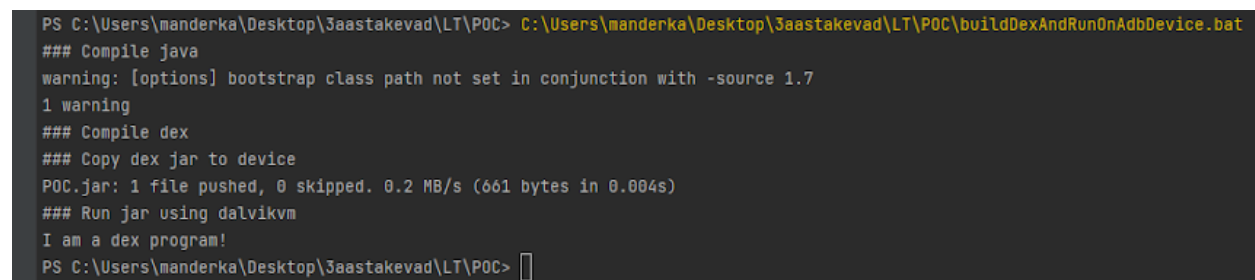
```
base=/data/local/tmp/HelloWorld
export CLASSPATH=$base/HelloWorld.jar
export ANDROID_DATA=$base
mkdir -p $base/dalvik-cache
exec app_process $base com.helloworld.HelloWorld "$@"
```

The script sets the necessary environment variables, such as CLASSPATH and ANDROID_DATA, creates directories for the base and Dalvik cache and executes the app_process command with

the path to the helloworld.jar file and the name of the main class as arguments. The `app_process` command is responsible for starting the Dalvik VM and running the Java program on Android. This approach allows developers to run Java 8 code on Android devices, even after the discontinued Dalvik VM [30].

3.1.6 Installation script

An installation script was created to streamline the execution of Java code on the Android device. The script was intended to simplify developing and deploying the POC Android application for which a Windows batch script was created (see Listing 1 in Appendix A). The script allowed the exploration of the inner workings of the Android operating System programmatically and was the foundation for upcoming POC malware development. Compiling Java code to byte code with IntelliJ compiler is the streamlined procedure for executing Java code on Android. The developed batch script is then utilised to DEX Java byte code. The required Dalvik structure is then generated and run in Android using the `run.sh` script (see Listing 2 in Appendix A). The `buildDexAndRunOnAdbDevice.bat` script is then executed, which outputs the program's output to the terminal (see Figure 15).



```
PS C:\Users\manderka\Desktop\Jaastakevad\LT\POC> C:\Users\manderka\Desktop\Jaastakevad\LT\POC\buildDexAndRunOnAdbDevice.bat
### Compile java
warning: [options] bootstrap class path not set in conjunction with -source 1.7
1 warning
### Compile dex
### Copy dex jar to device
POC.jar: 1 file pushed, 0 skipped. 0.2 MB/s (661 bytes in 0.004s)
### Run jar using dalvikvm
I am a dex program!
PS C:\Users\manderka\Desktop\Jaastakevad\LT\POC> 
```

Figure 15. The output of example code by `BuildDexAndRunOnAdbDevice.bat` script

The created script was utilised through the POC development phase and has saved countless hours by streamlining simple yet time-consuming tasks by hand.

memory offsets cannot be used [33]. The verdict is that memory manipulation is unsuitable for reliably extracting the Mobile-ID PINs from `Stk.apk` process. This result makes it pointless to continue finding ways to automatically confirm the Mobile-ID transactions through memory manipulation.

3.3 Patching Stk.apk

Patching in software terms means modifying an existing application's code by updating the program or decompiling, modifying the code, and then recompiling the code. The Androids STK application could be modified to contain the malware business logic directly inside the STK application. VisualStudios APKLab extension was used for patching the STK Android application [34]. The `Stk.apk` application at `/system/app/Stk/stk.apk` was copied from the smartphone device through SSH file manager and then loaded to APKLab (see Figure 17) with `decompile_java` and `-only-main-classes` options (see Figure 18). Initially, there was a problem with the lack of Android and XIAOMI frameworks at decompilation (see Listing 3 in Appendix A), but it was solved by pulling `miui.apk` and `framework-res.apk` from the device and installing them in APKLab before decompilation [35].

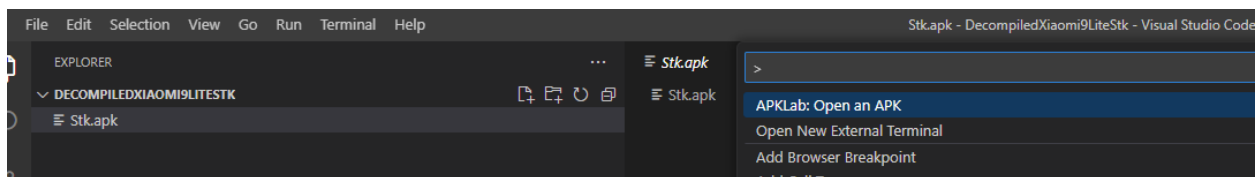


Figure 17. Loading the `Stk.apk` to APKLab

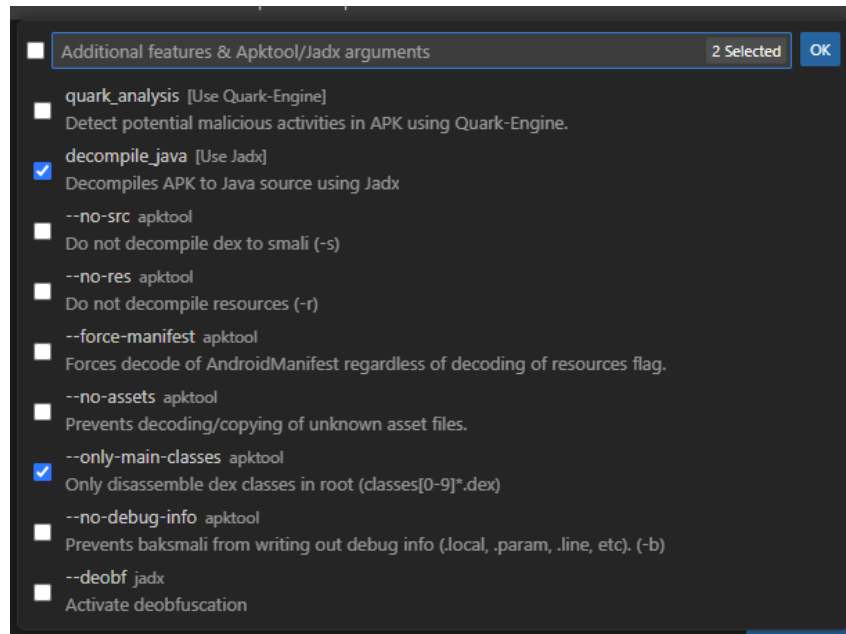


Figure 18. Options used for decompiling Stk.apk using APKLab

The result of decompiling Stk.apk is a decompiled application structure in Smali. Smali mixes assembly and human-readable language for the DEX format created to modify DEX code [29]. A small portion of the `StkInputActiviy.SendResponse` method was modified for testing purposes to record the entered digits (see Figure 19). Please note that Smali is used by APKLab for recompilation. Java is only used to make the code more readable to humans.

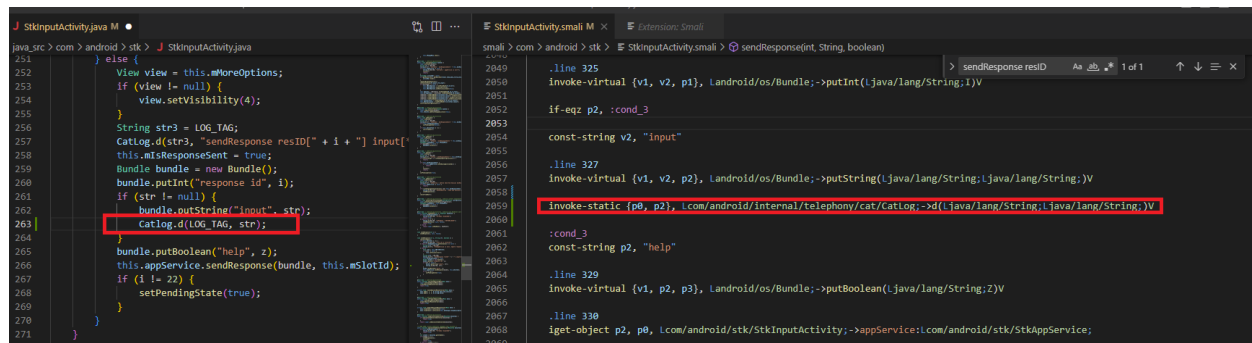
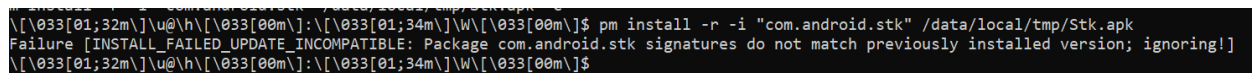


Figure 19. Modification of StkInputActiviy.SendResponse

The modified Smali code was reconstructed by right-clicking on the `apktool.yml` file and selecting “APKLab: Rebuild the APK”. This produced a rebuild log (see Listing 4 in Appendix A) and a modified Apk in the “dist” folder. However, there was a problem with getting the newly built Stk.apk on the device “/system/app/Stk” directory due to the system directory being

read-only. Using the command “`mount -o rw,remount /`”, the system directory was remounted with read-write permissions [36]. Additionally, an error occurred while attempting to install the reconstructed Stk.apk because the new Stk.apk was signed with a different key than the prior legitimate version of the Stk.apk (see Figure 20). Due to an internal error, attempts to uninstall the previous version of the Stk.apk failed. This issue was resolved by repeatedly executing the “`pm uninstall com.android.stk`” command at any available time until it succeeded at the end of a device reboot.



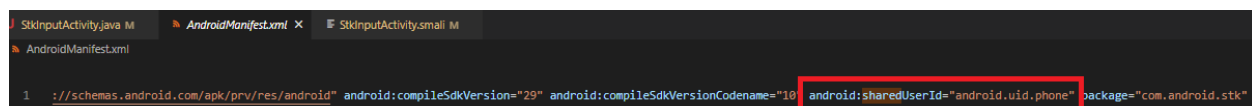
```

\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\W\[\033[00m\]$ pm install -r -i "com.android.stk" /data/local/tmp/Stk.apk
Failure [INSTALL_FAILED_UPDATE_INCOMPATIBLE: Package com.android.stk signatures do not match previously installed version; ignoring!]
\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\W\[\033[00m\]$

```

Figure 20. Package manager rejecting installation attempt due to signature mismatch

Even though the previous version of the Stk.apk was removed, the patched version could not be installed due to the Stk.apk having the same shared user id with the Telephony service (see Listing 5 in Appendix A). Android’s shared user id is used to give access to other apps resources with the same user id (see Figure 21) [37].



```

1  <?xml version="1.0" encoding="utf-8"?>
  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:compileSdkVersion="29" android:compileSdkVersionCodename="10"
    android:sharedUserId="android.uid.phone" package="com.android.stk">

```

Figure 21. Stk.apk AndroidManifest.xml with sharedUserId

The shared user ID was removed from Stk.apk AndroidManifest.xml and rebuilt, resulting in a semi-working application. LogCat logs indicated that the application was running but was missing the required permissions that it got from the shared user ID to receive STK commands from Android broadcasts (see Listing 6 in Appendix A). Android operating System uses Android broadcasts to transmit messages between different applications. There is a way to manually add permissions to Android applications by using the Android package manager `grant` command, but this resulted in a security exception that states that `RECEIVE_STK_COMMANDS` permission is not a changeable permission type (see Figure 22).


```

\\[033[01;32m\\]u@\\h[033[00m\\]:\\[033[01;34m\\]W[033[00m\\]$ pm grant com.android.stk android.permission.RECEIVE_STK_COMMANDS
Security exception: Permission android.permission.RECEIVE_STK_COMMANDS requested by com.android.stk is not a changeable permission type

java.lang.SecurityException: Permission android.permission.RECEIVE_STK_COMMANDS requested by com.android.stk is not a changeable permission type
    at com.android.server.pm.permission.BasePermission.enforceDeclaredUsedAndRuntimeOrDevelopment(BasePermission.java:429)
    at com.android.server.pm.permission.PermissionManagerService.grantRuntimePermission(PermissionManagerService.java:2134)
    at com.android.server.pm.permission.PermissionManagerService.access$900(PermissionManagerService.java:122)
    at com.android.server.pm.permission.PermissionManagerService$PermissionManagerServiceInternalImpl.grantRuntimePermission(PermissionManagerService.java:3088)
    at com.android.server.pm.PackageManagerService.grantRuntimePermission(PackageManagerService.java:5869)
    at com.android.server.pm.PackageManagerShellCommand.runGrantRevokePermission(PackageManagerShellCommand.java:1955)
    at com.android.server.pm.PackageManagerShellCommand.onCommand(PackageManagerShellCommand.java:230)
    at android.os.ShellCommand.exec(ShellCommand.java:104)
    at com.android.server.pm.PackageManagerService.onShellCommand(PackageManagerService.java:22451)
    at android.os.Binder.shellCommand(Binder.java:881)
    at android.os.Binder.onTransact(Binder.java:765)
    at android.content.pm.IPackageManager$Stub.onTransact(IPackageManager.java:4924)
    at com.android.server.pm.PackageManagerService.onTransact(PackageManagerService.java:4115)
    at android.os.Binder.execTransactInternal(Binder.java:1021)
    at android.os.Binder.execTransact(Binder.java:994)
\\[033[01;32m\\]u@\\h[033[00m\\]:\\[033[01;34m\\]W[033[00m\\]$

```

Figure 22. Package manager denying the granting of RECIEVE_STK_COMMANDS to com.android.stk application

Alternatively, these permissions can be manually granted by modifying the XML entry of “/data/system/packages.xml” com.android.stk package perms. Even though the permissions were added (see Listing 7 in Appendix A), they did not function because the package manager revoked them after the reboot (see Figure 23).

```

PackageManager: Un-granting permission android.permission.RECEIVE_STK_COMMANDS from package com.android.stk (protectionLevel=18 flags=0x2008be44)
PackageManager: Un-granting permission android.permission.SET_ACTIVITY_WATCHER from package com.android.stk (protectionLevel=2 flags=0x2008be44)
PackageManager: Un-granting permission android.permission.START_ACTIVITIES_FROM_BACKGROUND from package com.android.stk (protectionLevel=49682 flags=0x2008be44)
PackageManager: Un-granting permission android.permission.USER_ACTIVITY from package com.android.stk (protectionLevel=18 flags=0x2008be44)

```

Figure 23. Package manager removing manually added permissions after the restart

The verdict is that it was not possible to modify Android system APKs due to the robust permission management system.

3.4 JAttach

JAttach, the JVM Dynamic Attach utility, enables the transmission of commands to a JVM process via the Dynamic Attach mechanism. It integrates the functionality of jmap, jstack, jcmd, and jinfo into a single program and can be used without installing a JDK, requiring only a JRE [38]. Even though the project’s GitHub page clearly states that this tool was designed for Java VM, considerable time was spent attempting to get JAttach to function on an Android device that uses Dalvik VM. The conclusion is that Android devices do not support the program, and the author of JAttach confirms it [39] .

3.5 Android Interface Definition Language (AIDL)

Android developers can create interfaces for various application components using the powerful Android Interface Definition Language (AIDL) tool. AIDL is very useful for communicating across several processes or apps. Developers may call methods on objects running in a different process or device using AIDL, an RPC technique. A client and server can exchange information by marshalling arguments and returning values even if they are executing on separate threads. Data types in AIDL span from simple integers, booleans, and floats to sophisticated lists and hashmaps. The callback feature allows one component to register a method that another can invoke. The Android SDK includes the “aidl” tool, which generates Java interfaces from AIDL files. However, before utilising the interface, developers must create an AIDL file containing the method signatures and data types. Lastly, executing the “aidl” tool on the AIDL file, which generates a Java interface that can be used in their application [40] . Telephony service AIDL-provided methods were investigated. This was done by loading `android.os.ServiceManager` from Javas dynamic class loader (see Listing 9 in Appendix A) [41]. `ServiceManager` with `ITelephony.aidl` (see Listing 8 in Appendix A) file resulted in a Binder interface that enabled it to interact with the Telephony service. The objective was to locate a specific method of Telephony service that returned a direct instance of `TelephonyManager` class. Java reflection would have been utilized to intercept STK commands inside of the `TelephonyManager` class, but unfortunately, Android’s provided “`ITelephony.aidl`” file lacks suitable methods for returning a `TelephonyManager` instance [42].

3.6 Radio Interface Layer Daemon (RILD)

Intercepting Radio Interface Layer Daemon (RILD) communication with Android’s Radio Interface Layer (RIL) is another method to intercept SIM commands (see Figure 24) [43].

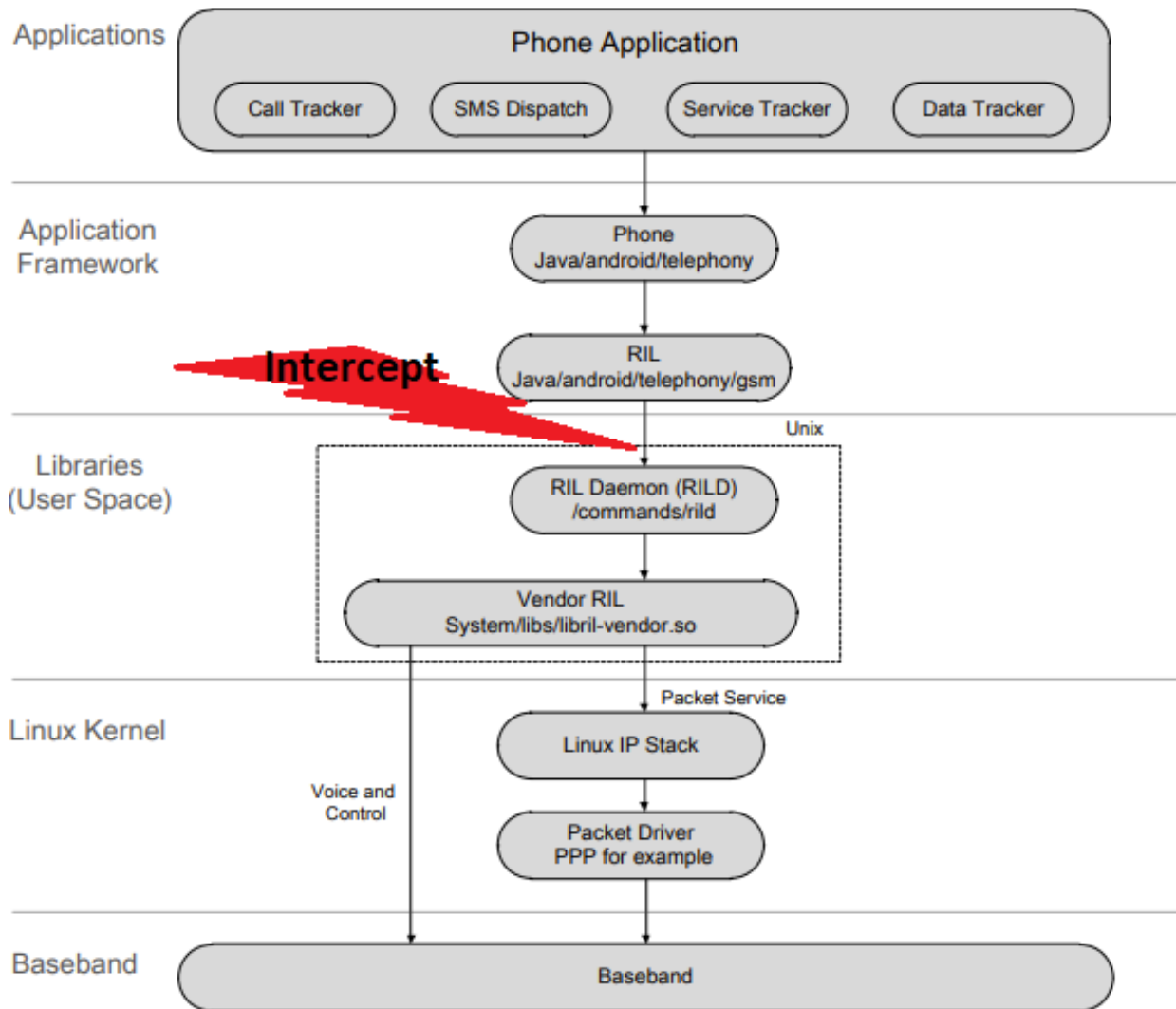


Figure 24. Diagram of Android RIL architecture with possible interception point [44]

According to the StackOverflow post, RILD socket would reside in the `/dev/socket` folder named “rild”, but this was not the case with the smartphone [43]. The StackOverflow post is quite old, but newer articles and programs support the `/dev/socket/rild` socket location [45] [46]. Filesystem paths such as `/run/socket`, `/data/misc`, `/data/system`, `/data/property`, `/dev`, and `/tmp` were also searched for the RILD socket, but the socket was not found. The conclusion is that the RILD socket was not discovered on the XIAOMI MI 9 lite smartphone, making it impossible to intercept the RILD traffic.

3.7 Binder

Android operating system employs Binders as a lightweight remote procedure call mechanism [47]. The Stk.apk Binder communication was looked into using the JTrace program. JTrace is a tool that offers system call tracing for ARM64 and x86 64 architectures and is especially useful for Android-aware tracing. It offers several advantages over the standard "strace" command, including colourized output, thread name support, and plugin architecture. JTrace can also decode socket-based I/O, system property settings, input events, and even Binder messages [48]. Analysing the StkAppService STK command communication flow indicates that SIMs commands are transmitted through the binder service only once (see Figure 5). Every other STK command transfer between StkAppService and Telephony service is conducted via direct callbacks. This eradicates the possibility of intercepting STK command results in Telephony and STK service command flow messages through the Binder service.

4. Discussion

Android-based malware that can exploit Mobile-IDs unencrypted STK command communication is theoretically conceivable. Due to the complexity of the Mobile-ID platform and the Android operating system, creating such malware would require significant effort. One potential avenue for future research could involve the development of a functional proof-of-concept (POC) malware. The Mobile-ID authentication flow is susceptible to exploitation through various Android processes. It is theoretically possible to develop MITM (Man-In-The-Middle) malware at the socket level for RILD (Radio Interface Layer Daemon) communication. The feasibility of injecting Java code into STK command handling-related processes has been established. However, the intricate nature of Android's code renders the process considerably complex. It remains possible to patch the Stk.apk through the identification of a means to authorise permissions associated with STK command broadcasting or by disabling the signature verification of the Androids Stk.apk application. The malware's feature set may include the ability to execute remote code, monitor Mobile-ID usage to create a dataset of compromised users, and create a proxy network using all compromised devices, making it harder to find the bad actor's actual connection location.

Aside from the development of POC malware, ensuring the malware's stable functionality across Android manufacturer's devices may be tricky. Each manufacturer applies unique modifications to their Android operating system, making it challenging to ensure the reliability of malware across the majority of presently used devices. This means that the Mobile-ID authentication process should be exploited at a universally standardised level across all manufacturers. Detecting malware in Mobile-ID authentication can pose significant challenges due to the diverse range of methods that can be utilised to exploit the authentication flow. The act of scanning Android operating system processes for injection points has the potential to identify processes that have been compromised. The detection of patching attempts can be achieved by comparing system process signatures with manufacturing signatures. Monitoring the duration of the authentication process has the potential to reveal instances of poorly implemented malware that engages in the authentication within a timeframe of less than one second, which may indicate improper timing. The creation of this detection is feasible at the level of the SIM card. Additional methods for detecting malware involve actively monitoring STK dialog creation commands and verifying the presence of the authentication dialogs. Verification of RILD layer manipulation by monitoring the

processes connected to the RIL socket could prevent RIL MITM attacks. A potential approach involves training a machine learning algorithm using legitimate and fraudulent authentication process logs. It is certain that the detection of malware is feasible owing to the comprehensive logging features of the Android operating system. In order to reduce the likelihood of Mobile-ID malware infection, smartphone users should prioritize the maintenance of their device's security updates, exercise caution when downloading applications from Google Play, abstain from connecting to public Wi-Fi networks to avoid potential exploitation of Android services or tricking into downloading malware application through man-in-the-middle (MITM) attacks and consider installing anti-virus software to conduct periodic scans of their device.

5. Conclusion

This study has explored the viability of taking control over the Mobile-ID authentication process using various methods. The main goal of the Android Mobile-ID SIM Toolkit command intercepting malware is to obtain Mobile-ID PINs, which can then be used for unauthorised authentication of Mobile-ID transactions. The research covers various techniques employed to investigate unauthorised entry. According to the study's results, the emergence of malicious software is a possibility. However, due to the complex code architecture of the Android operating system and the implemented security features, no such software was ultimately successful. Despite the absence of a POC application, the potential threat posed by this type of malware persists. Furthermore, the identification of this type of malware is expected to pose a challenge owing to its execution with root privileges. It is essential for Android phone users to consistently update their devices as a precautionary measure.

References

- [1] Estonian Government, “Estonian ID software,” [Online]. Available: <https://www.id.ee/en/>. [Accessed 1 5 2023].
- [2] Estonian Government, “Estonian Mobile-ID,” [Online]. Available: <https://www.id.ee/en/mobile-id/>. [Accessed 1 5 2023].
- [3] B. Vuleta, “44 Worrying Malware Statistics to Take Seriously in 2023,” 30 3 2023. [Online]. Available: <https://legaljobs.io/blog/malware-statistics/>. [Accessed 1 5 2023].
- [4] D. Curry, “Android Statistics (2023) - Business of Apps,” 27 2 2023. [Online]. Available: <https://www.businessofapps.com/data/android-statistics/>. [Accessed 1 5 2023].
- [5] Estonian Government, “ID-card as an identity document,” [Online]. Available: <https://www.id.ee/en/article/id-card-as-an-identity-document/>. [Accessed 1 5 2023].
- [6] Estonian Government, “Estonian ID Introduction,” [Online]. Available: <https://www.id.ee/en/rubriik/introduction/>. [Accessed 1 5 2023].
- [7] K. Valgur, “Twenty years ago, the first ID card was issued (in Estonian),” 22 1 2022. [Online]. Available: <https://arileht.delfi.ee/artikkel/95755585/kakskummend-aastat-tagasi-valjastati-esimene-id-kaart>. [Accessed 1 5 2023].
- [8] Estonian Government, “Mobile-ID: digital identity document on smart phone,” [Online]. Available: <https://www.id.ee/en/article/mobile-id-digital-identity-document-on-smart-phone/>. [Accessed 1 5 2022].
- [9] Generation Partnership Project (3GPP), “Specification of the SIM Application Toolkit (SAT) for the Subscriber Identity Module - Mobile Equipment (SIM-ME) interface,” 6 2007. [Online]. Available:

https://www.etsi.org/deliver/etsi_ts/101200_101299/101267/08.18.00_60/ts_101267v081800p.pdf. [Accessed 1 5 2023].

- [10] S. Kravtšenko, “The Estonian Mobile-ID Implementation on the SIM Card. BSc thesis, Institute of Computer Science of University of Tartu,” 2022. [Online]. Available: https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=74566&year=2022&language=en. [Accessed 1 5 2022].
- [11] Google, “StkAppService.java - Android Code Search,” [Online]. Available: <https://cs.android.com/android/platform/superproject/+/refs/heads/master:packages/apps/Stk/src/com/android/stk/StkAppService.java>. [Accessed 4 5 2023].
- [12] Google, “CatService.java - Android Code Search,” [Online]. Available: <https://cs.android.com/android/platform/superproject/+/refs/heads/master:frameworks/opt/telephony/src/java/com/android/internal/telephony/cat/CatService.java>. [Accessed 1 5 2023].
- [13] Google, “StkAppService.java - Android Code Search,” [Online]. Available: <https://cs.android.com/android/platform/superproject/+/refs/heads/master:packages/apps/Stk/src/com/android/stk/StkAppService.java;l=1065;drc=7346c436e5a11ce08f6a80dcfeb8ef941ca30176;bpv=1;bpt=1?q=StkAppSe&ss=android%2Fplatform%2Fsuperproject>. [Accessed 1 5 2023].
- [14] Google, “StkAppService.java - Android Code Search,” [Online]. Available: <https://cs.android.com/android/platform/superproject/+/refs/heads/master:packages/apps/Stk/src/com/android/stk/StkAppService.java;l=1277;drc=7346c436e5a11ce08f6a80dcfeb8ef941ca30176>. [Accessed 1 5 2023].
- [15] J. Ruge, “Critical Bluetooth Vulnerability in Android (CVE-2020-0022) – BlueFrag,” 6 2 2020. [Online]. Available: <https://insinuator.net/2020/02/critical-bluetooth-vulnerability-in-android-cve-2020-0022/>. [Accessed 4 5 2023].

- [16] M. Humphries, “36 Malicious Android Apps Found on Google Play, Did You Install Them?,” 27 7 2022. [Online]. Available: <https://www.pcmag.com/news/36-malicious-android-apps-found-on-google-play-did-you-install-them>. [Accessed 4 5 2023].
- [17] Federal Communication Commission, “What is 'Juice Jacking' and Tips to Avoid It,” 27 4 2023. [Online]. Available: <https://www.fcc.gov/juice-jacking-tips-to-avoid-it>. [Accessed 4 5 2023].
- [18] Estonian Government, “Smart-ID,” [Online]. Available: <https://www.smart-id.com>. [Accessed 4 5 2023].
- [19] S. Maala, “A Proof of Concept Malware for Interacting with the Smart-ID Android Application. BSc thesis, Institute of Computer Science of University of Tartu,” 2020. [Online]. Available: https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=69678&year=2020&language=en. [Accessed 1 5 2023].
- [20] Z. Aayush, “What is Android Rooting?,” 2 10 2022. [Online]. Available: <https://www.geeksforgeeks.org/what-is-android-rooting/>. [Accessed 1 5 2023].
- [21] csyuma, “[HELP!!!!] Huawei unlock code,” 26 3 2019. [Online]. Available: <https://forum.xda-developers.com/t/help-huawei-unlock-code.3915218/>. [Accessed 1 5 2023].
- [22] M. Lee, “How to Root Xiaomi Mi 9!,” 3 6 2019. [Online]. Available: <https://www.youtube.com/watch?v=e2lfbAXoUYA>. [Accessed 1 5 2023].
- [23] Google, “Titanium Backup (root needed),” [Online]. Available: https://play.google.com/store/apps/details?id=com.keramidas.TitaniumBackup&hl=en_US. [Accessed 1 5 2023].

- [24] Android, “Android Debug Bridge (adb),” [Online]. Available: <https://developer.android.com/tools/adb>. [Accessed 1 5 2023].
- [25] Google, “SSHelper,” [Online]. Available: <https://play.google.com/store/apps/details?id=com.arachnoid.sshelper>. [Accessed 1 5 2023].
- [26] Bitvise, “Bitvise SSH Client: Free SSH file transfer, terminal and tunneling,” [Online]. Available: <https://www.bitvise.com/ssh-client>. [Accessed 1 5 2023].
- [27] Jelonmusk, “Difference Between Dalvik and ART in Android,” 3 1 2023. [Online]. Available: <https://www.geeksforgeeks.org/difference-between-dalvik-and-art-in-android/>. [Accessed 1 5 2023].
- [28] Android, “d8,” 12 4 2023. [Online]. Available: <https://developer.android.com/tools/d8>. [Accessed 1 5 2023].
- [29] TheMobileSecurityGuys, “Smali: Assembler for Android’s VM,” 9 6 2020. [Online]. Available: <https://mobsecguys.medium.com/smali-assembler-for-dalvik-e37c8eed22f9>. [Accessed 1 5 2023].
- [30] P. Ahlbrecht, “How to run Java programs directly on Android (without creating an APK),” 5 10 2020. [Online]. Available: <https://raccoon.onyxbits.de/blog/run-java-app-android>. [Accessed 1 5 2023].
- [31] E. Heijnen, “Cheat Engine,” [Online]. Available: <https://www.cheatengine.org/aboutce.php>. [Accessed 1 5 2023].
- [32] 電腦ケロちゃん, “Dumproid,” [Online]. Available: <https://github.com/tkmru/dumproid>. [Accessed 1 5 2023].

- [33] Oracle, “The Structure of the Java Virtual Machine,” 3 2014. [Online]. Available: <https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5.3>. [Accessed 1 5 2023].
- [34] APKLab, “APKLab,” [Online]. Available: <https://github.com/APKLab/APKLab>. [Accessed 1 5 2023].
- [35] uiop_uiop_uiop, “Can’t find framework resources for package of id: 18. You must install proper framework files,” 31 5 2022. [Online]. Available: https://blog.csdn.net/uiop_uiop_uiop/article/details/125057458. [Accessed 1 5 2023].
- [36] N. M. Ragib, “adb remount fails - mount: 'system' not in /proc/mounts,” 6 3 2019. [Online]. Available: <https://stackoverflow.com/questions/55030788/adb-remount-fails-mount-system-not-in-proc-mounts>. [Accessed 1 5 2023].
- [37] Android, “App manifest file,” [Online]. Available: <https://developer.android.com/guide/topics/manifest/manifest-element#uid>. [Accessed 1 5 2023].
- [38] JAttach, “JAttach,” [Online]. Available: <https://github.com/jattach/jattach>. [Accessed 1 5 2023].
- [39] jattach, “Android aarch64 not working,” [Online]. Available: <https://github.com/jattach/jattach/issues/69>. [Accessed 5 5 2023].
- [40] Android, “Android Interface Definition Language (AIDL),” [Online]. Available: <https://developer.android.com/guide/components/aidl>. [Accessed 1 5 2023].
- [41] Coding Blocks, “Getting the instance of a system service without an application context in Android,” 29 8 2017. [Online]. Available: <https://blog.codingblocks.com/2017/getting-the-instance-of-a-system-service-without-an-application-context-in-android/>. [Accessed 1 5 2023].

- [42] Google, “ITelephony.aidl - Android Code Search,” [Online]. Available: <https://cs.android.com/android/platform/superproject/+/master:frameworks/base/telephony/java/com/android/internal/telephony/ITelephony.aidl?q=interface%20ITelephony%20%7B&ss=android%2Fplatform%2Fsuperproject>. [Accessed 1 5 2023].
- [43] t0mm13b, “How does Modem code talk to Android code,” 6 4 2013. [Online]. Available: <https://stackoverflow.com/questions/11111067/how-does-modem-code-talk-to-android-code>. [Accessed 1 5 2023].
- [44] Quectel, “Android RIL Driver,” 19 4 2021. [Online]. Available: https://www.quectel.com/wp-content/uploads/2021/03/Quectel_Android_RIL_Driver_User_Guide_V2.0-1.pdf. [Accessed 1 5 2023].
- [45] L. Clay, “How to use rild command,” 25 7 2022. [Online]. Available: <https://copyprogramming.com/howto/how-to-use-rild-command>. [Accessed 1 5 2023].
- [46] Componolit, “RILProxy,” [Online]. Available: <https://github.com/Componolit/rilproxy>. [Accessed 5 5 2023].
- [47] Google, “Binder,” [Online]. Available: <https://developer.android.com/reference/android/os/Binder>. [Accessed 5 5 2023].
- [48] J. Levin, “JTrace,” 12 5 2021. [Online]. Available: <http://newandroidbook.com/tools/jtrace.html>. [Accessed 1 5 2023].
- [49] TheMobileSecurityGuys, “Medium,” 9 6 2020. [Online]. Available: <https://mobsecguys.medium.com/smali-assembler-for-dalvik-e37c8eed22f9> . [Accessed 1 5 2023].

- [50] S. Hazarika, “Most popular custom ROMs for Android in 2023,” 24 3 2023. [Online]. Available: <https://www.xda-developers.com/most-popular-custom-roms-android/>. [Accessed 1 5 2023].
- [51] C. Stouffer, “Bloatware: What it is + how to spot and remove it,” 28 11 2022. [Online]. Available: <https://us.norton.com/blog/online-scams/bloatware>. [Accessed 1 5 2023].
- [52] NortonLifeLockEmployee, “The risks of rooting your Android phone,” 21 9 2022. [Online]. Available: <https://us.norton.com/blog/mobile/android-rooting-risks>. [Accessed 1 5 2023].

Appendices

A Code and log snippets

Listing 1. Windows batch script “compileDexFromClassAndRunOnAdbDevice.bat” for the streamlined running of Java code on Android devices

```
@echo off
echo ### Compile POC dex
call "%cd%\sdk\build-tools\30.0.3\d8.bat" --output=POC.jar --lib
"%cd%\sdk\platforms\android-33\android.jar" --classpath "%cd%\out\production\POC"
"%cd%\out\production\POC\com\poc\POC.class"
echo ### Copy POC to device and set it up
call "%cd%\sdk\platform-tools\adb.exe" shell su -c mkdir -p /data/local/tmp/POC
call "%cd%\sdk\platform-tools\adb.exe" push POC.jar /storage/emulated/0/POC.jar
call "%cd%\sdk\platform-tools\adb.exe" shell su -c mv /storage/emulated/0/POC.jar
/data/local/tmp/POC/POC.jar
call "%cd%\sdk\platform-tools\adb.exe" push run.sh /data/local/tmp/POC/run.sh
call "%cd%\sdk\platform-tools\adb.exe" shell su -c chmod 777
/data/local/tmp/POC/run.sh
echo ### Run jar using run.sh
call "%cd%\sdk\platform-tools\adb.exe" shell su -c ./data/local/tmp/POC/run.sh
```

Listing 2. Bash script “run.sh” for starting DEX application on Android devices

```
base=/data/local/tmp/POC
export CLASSPATH=$base/POC.jar
export ANDROID_DATA=$base
mkdir -p $base/dalvik-cache
exec app_process $base com.poc.POC "$@"
```

Listing 3. Visual Studio APKLab extension decompilation log

```

-----
Decoding Stk.apk into c:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk
-----
java -jar C:\Users\manderka\apklab\apktool_2.7.0.jar d c:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk.apk -o
c:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk --only-main-classes
I: Using Apktool 2.7.0 on Stk.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\manderka\AppData\Local\apktool\framework\1.apk
W: Could not decode attr value, using undecoded value instead: ns=android, name=theme, value=0x100d00c5
W: Could not decode attr value, using undecoded value instead: ns=android, name=theme, value=0x100d00c5
W: Could not decode attr value, using undecoded value instead: ns=android, name=theme, value=0x100d00c5
W: Could not decode attr value, using undecoded value instead: ns=android, name=theme, value=0x100d00c6
W: Could not decode attr value, using undecoded value instead: ns=android, name=theme, value=0x100d00c6
I: Regular manifest package...
I: Decoding file-resources...
W: Could not decode attr value, using undecoded value instead: ns=, name=style, value=0x100d004d
W: Could not decode attr value, using undecoded value instead: ns=android, name=textAppearance, value=0x100d005f
I: Decoding values */* XMLs...
Can't find framework resources for package of id: 16. You must install proper framework files, see project website for more info.
Decoding process exited with code 1
-----
Initializing c:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk as Git repository
-----
cd /d "c:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk" && git init && git config core.safecrlf false && git add -A && git commit -q -m
"Initial APKLab project"
Initialized empty Git repository in C:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk\.git\
Initializing Git process was successful

```

Listing 4. Visual Studio APKLab extension rebuild log

```

-----
Rebuilding Stk.apk into Stk1\dist
-----
java -jar C:\Users\manderka\apklab\apktool_2.7.0.jar b c:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk1 --use-aapt2
I: Using Apktool 2.7.0
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk into: c:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk1\dist\Stk.apk
Rebuilding process was successful
-----
Signing Stk1\dist\Stk.apk
-----
java -jar C:\Users\manderka\apklab\uber-apk-signer-1.2.1.jar -a
c:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk1\dist\Stk.apk --allowResign --overwrite
source:
C:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk1\dist
zipalign location: BUILT_IN
C:\Users\manderka\AppData\Local\Temp\uapksigner-7702985726426611915\win-
zipalign_29_0_2.exe7046064833938616271.tmp
keystore:
[0] 390ab2d2 C:\Users\manderka\.android\debug.keystore (DEBUG_ANDROID_FOLDER)
01. Stk.apk
SIGN
file: C:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk1\dist\Stk.apk (0.27 MiB)
checksum: 7338594b8e0293d14c5e21addb2614344cbbbb81886ae03abbc3bf08015495dee (sha256)
- zipalign success

```


- sign success

```
VERIFY
file: C:\Users\manderka\Desktop\3aastakevad\LT\STK\Stk1\dist\Stk.apk (0.28 MiB)
checksum: 86bf8a8d833997e9dee19b734e29cacb1c2359263ecbf0654153bb0af1dfe6a3 (sha256)
- zipalign verified
- signature verified [v3]
  Subject: C=US, O=Android, CN=Android Debug
  SHA256: 0978a2fbf4bdfc7d2ce0000c97a6574fc2205f87b05077599b1bb35f84f91d42 / SHA256withRSA
Expires: Fri Feb 21 15:50:56 EET 2053
[Tue Mar 28 19:49:19 EEST 2023][v1.2.1]
Successfully processed 1 APKs and 0 errors in 0.62 seconds.
```

Signing process was successful

Listing 5. Visual Studio APKLab extension rebuild log

```
pm install /system/app/Stk/Stk.apk
Failure [INSTALL_FAILED_SHARED_USER_INCOMPATIBLE: Reconciliation failed...: Reconcile failed: Package
com.android.stk has no signatures that match those in shared user android.uid.phone; ignoring!]
```

Listing 6. Patched Stk.apk without `sharedUserId` receiving permission denied error for STK command broadcasts

```
03-29 14:27:08.253 1586 1727 W BroadcastQueue: Permission Denial: receiving Intent {
act=com.android.internal.stk.command flg=0x10000010 cmp=com.android.stk/.StkCmdReceiver (has extras) } to
com.android.stk/.StkCmdReceiver requires android.permission.RECEIVE_STK_COMMANDS due to sender
com.android.phone (uid 1001)
```

Listing 7. Required permissions for `com.android.stk`

```
<item name="android.permission.RECEIVE_STK_COMMANDS" granted="true" flags="0" />
<item name="android.permission.SET_ACTIVITY_WATCHER" granted="true" flags="0" />
<item name="android.permission.START_ACTIVITIES_FROM_BACKGROUND" granted="true" flags="0" />
<item name="android.permission.USER_ACTIVITY" granted="true" flags="0" />
```

Listing 8. AIDL code that exposes Telephony services dial method

```
package com.android.internal.telephony;

interface ITelephony {
    void dial(String number);
}
```

Listing 9. Java code that utilises AIDL generated Telephony service with example dial method

```
package com.poc;

import android.os.IBinder;
import android.os.RemoteException;
import com.android.internal.telephony.ITelephony;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class POC {
    public static void main(String[] args) throws ClassNotFoundException, NoSuchMethodException,
    InvocationTargetException, IllegalAccessException, RemoteException {
        Method getServiceMethod =
        Class.forName("android.os.ServiceManager").getDeclaredMethod("getService", String.class);
        IBinder phoneBinder = (IBinder) getServiceMethod.invoke(null, new
        Object[]{"phone"});
        ITelephony telephony = ITelephony.Stub.asInterface(phoneBinder);
        // example use of telephony interface
        telephony.dial("+372 12345678");
    }
}
```

B Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Karl Erik Mander,

(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Intercepting Mobile-ID SIM Toolkit Calls On Android,

(title of thesis)

supervised by Arnis Paršovs.

(supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Karl Erik Mander

2023-05-01