UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Silver Maala

A Proof of Concept Malware for Interacting with the Smart-ID Android Application

Bachelor's Thesis (9 ECTS)

Supervisor: Arnis Paršovs, MSc

Tartu 2020

A Proof of Concept Malware for Interacting with the Smart-ID Android Application

Abstract:

The aim of this thesis is to study how a malicious application can interact with the Smart-ID Android application. The result of this paper is a proof of concept application that is able to use root privileges to capture the PINs entered by the user in the Smart-ID transaction screen and is later able to automatically enter the captured PINs in the transaction screen.

Keywords:

Smart-ID, Android, malware, proof of concept

CERCS:

P170 - Computer science, numerical analysis, systems, control

Kontseptuaalne pahavara Androidi Smart-ID rakendusele

Lühikokkuvõte:

Käesoleva töö eesmärk on uurida kuidas pahavara saab mõjutada Smart-ID rakendust Androidis. Töö tulemuseks on kontseptuaalne pahavara rakendus, mis kasutab root õigusi, et kinni püüda kasutaja sisestatud PIN koode Smart-ID rakenduses ja hiljem sisestada kinni püütud PIN koode automaatselt.

Võtmesõnad:

Smart-ID, Android, pahavara, kontseptuaalne tarkvara

CERCS:

P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1. Introduction

Smart-ID is a popular smartphone application in Estonia. It is used for authenticating oneself online in various places, including banks and government e-services [1], and for digitally signing documents. In fact, a signature done with Smart-ID is equal to a handwritten signature [2, 3].

Although the security of the Smart-ID cryptography has been written about in “Server-Supported RSA Signatures for Mobile Devices” by Buldas, Kalu, Laud and Oruaas in 2017 [4], currently there is no research into the security of the Smart-ID application.

The aim of this thesis is to study how a malicious application can interact with the Smart-ID application on an Android device where it has root privileges. The goal of the malicious app is to learn the Smart-ID PINs and to later automatically enter them into the Smart-ID app. The result of this thesis is a proof of concept (PoC) application for Android that works with the latest version of the Smart-ID app as of writing this thesis (18.3.175). The proof of concept application is able to capture the user’s inputs, learn the PINs and input the PINs automatically, using the tools available in Android.

The thesis is organized as follows. The first section introduces Smart-ID and gives a basic overview on what it is, how it works and looks into its cryptographic and software protection. The second section focuses on Android. A brief overview is given on Android security that looks into what prevents a malicious application from achieving its goals and additionally, we give glimpse into the Android ecosystem and why malware is such a threat for Android devices. The third section explains the creation of the malicious app. The first part details the testing and development environment setup and the second part describes the proof of concept application and how it achieves its goal: how it captures the PINs and enters them in the Smart-ID application. The final section details the code and logic of the proof of concept application, as well as what additional features are needed for it to be used in the real world.

2. Smart-ID

Smart-ID is a mobile application for iOS and Android created by SK ID Solutions and Cybernetica. It is a popular application, having 2 601 325 total users in December 2019 and 49 547 250 transactions in November 2019 [2].

With the Smart-ID application a user can authenticate themselves online, sign documents, use e-services and do online banking. A signature made with the Smart-ID application is legally binding and equal to a hand written signature. In addition, a signature made with this application is accepted by every EU member state, since Smart-ID is recognized as a QSCD (Qualified Signature Creation Device) [2].

2.1. Cryptography

Smart-ID is protected by a unique cryptographic method. The paper “Server-Supported RSA Signatures for Mobile Devices” by Buldas, Kalu, Laud and Oruaas [4] explains the cryptography Smart-ID uses in detail, but in this section we give a brief summary.

Smart-ID cryptographic method employs both a server and a mobile device. Here both the server and the mobile device hold a part of the private key. Furthermore, both generate their parts of the key independently of one another so neither of them ever have the full private key. This means that to get the correct private key, the device and the server have to work together.

During the authentication process the PIN is used to decrypt the user’s part of the RSA key stored on the device. The user’s part of the private key is used to generate a share of the signature and then the share of the signature is sent to the server. The server uses its share of the RSA key to complete the signature. If the signature can be verified, then the user entered the correct PIN. If not the user entered an incorrect PIN.

This method offers multiple advantages from a security standpoint. If the private key is held only on the device an attacker could extract it from memory and forge the user’s signature. Even if the private key is encrypted with the user’s password, those are not always secure enough and may be breached with a dictionary attack.

With this method, even if the user's share of the key is cloned, the attacker still needs the other part stored on the server and for that they need to communicate with the server. If the key is encrypted with the user's PIN, then a brute force and dictionary attacks are not possible since the attacker cannot know whether the resulting key is correct or not without the server and the server only allows 3 attempts before locking the account.

2.2. Authenticating and Signing

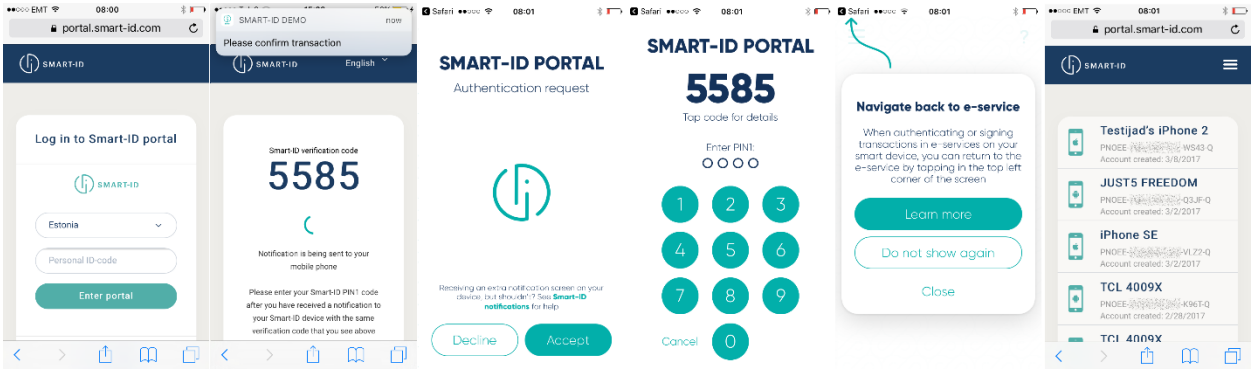

Figure 1. Process of authenticating using Smart-ID [5].

The process of authenticating and signing in the Smart-ID application is straight forward. The steps the user has to do are shown in Figure 1. In addition, the Smart-ID's technical documentation explains the process of authenticating in great detail. Below is a simplified summary of the authenticating process from the Smart-ID's technical documentation [6].

1. User goes to a website that supports authentication with Smart-ID and selects that option. When prompted they enter their user identifier, which could be their personal code or username etc.

2. The website sends an authentication request to the Smart-ID server and after receiving the response the website displays the Verification Code. The same Verification Code is later displayed on the Smart-ID transaction screen on the user's device. This lets the user verify that the PIN is asked for the correct transaction.

4. Smart-ID server sends a notification to the Smart-ID application on the user's device. There the Smart-ID application displays the Verification Code and asks for the PIN.

Which PIN is asked depends on the type of transaction: PIN1 is used for authenticating, PIN2 for signing.

5. User enters the PIN. The Smart-ID application generates part of the signature using its part of the private key and sends it to the Smart-ID server

6. The Smart-ID server completes the signature generation process with its the shares of the authentication key pair.

7. The website requests the authentication status from the Smart-ID server and the server sends a response.

8. The website confirms that the response is valid and creates a new session with the user's identity.

2.3. Software Protection

To prevent unauthorized use of the Smart-ID app, many protections have been put in place. The major protective features are listed below.

- The Smart-ID application has cloning protection [6].
- We observed that by default the user cannot take screenshots in the Smart-ID application, although it is possible to disable this protection in the application settings. This most likely means that the Smart-ID application has enabled FLAG_SECURE. This flag prevents the user from taking screenshots [7].
- The Smart-ID website states that Smart-ID needs to be up to date or else it will not work due to security risks [8].
- The Smart-ID website states that the Smart-ID application will not work on an outdated OS [9].
- Smart-ID checks if the device is rooted [10]. When creating a new account on the device Smart-ID gave us a notice, that the device was rooted.
- To prevent brute-force attacks if the PIN is entered incorrectly three times the account is locked for either 3 or 24 hours. If the account is locked three times the

account is disabled and the certificates revoked. After that the user needs to create a new account [11].

- Some websites set extra security for signing with Smart-ID. In such a case the Smart-ID app displays three Verification Codes among which the user has to select the one shown in the e-service [12]. This feature prevents phishing attacks where an attacker initiates the Smart-ID transaction and the user enters the PIN without checking the Verification Code.

3. Android

Android is an open-source operating system, mainly designed for touchscreen mobile devices. Currently it is the world's most popular OS, with almost 40% of the market share [13] and is used in more than 2.5 billion devices [14].

3.1. Security

Android is based on the Linux kernel, specifically Security-Enhanced Linux (SELinux) with many security enhancements [15]. All software running above the kernel run within the Application Sandbox [16]. The Application Sandbox is where all applications run as a unique user with its own unique UID [16] and where all applications, their memory, storage, resources are isolated from one-another [15]. This means that by default applications cannot interact with each other in any way. This prevents malicious code from interacting with other apps. The only way to access another application, its files or some functionality without exploiting flaws in the OS is through content providers [17], but this specific functionality has to be added in by the developers.

Furthermore, a lot of functionality is locked behind permissions. Permissions prevent apps from accessing functionality they are not supposed to access and prevent them from performing any actions that would affect the system, user or other apps [18]. For example, sensitive API's like the camera, GPS etc. can only be accessed through the OS if given the permission. But even with all the permissions an application still cannot interact with another application.

For malware to access another app and its related files the malicious application would need to break out of the Application Sandbox into the kernel [16]. It could do that by exploiting some vulnerability in the OS to escalate its privileges to root. With root access the malware would have full control of the device and can access, read and modify all the files in the system.

3.2. Android Vulnerabilities

The scenario of malware exploiting a vulnerability in Android to get root privileges is quite likely, considering the fact that when all the known vulnerabilities were tallied up, Android had the most vulnerabilities of any OS in 2016, 2017 and 2019. In fact, in 2019 Android had 414 vulnerabilities, 54 vulnerabilities more than Debian Linux in second place [19].

Unfortunately, patching vulnerabilities in Android is not straightforward. Many smartphone manufacturers have their own modified versions of Android, for example, Samsung has One UI [20] and OnePlus has Oxygenos [21], and it might take time for updates to reach these modified versions of Android. Two researchers, Nohl and Lell [22], showed that different phone manufacturers handle patching differently, sometimes even missing patches. Not to mention that the manufactures' own versions of Android handle security differently and can have their own vulnerabilities.

This is assuming the users even install or get these patches. Research by Duo Security [23], a cloud-based security provider, discovered that in 2016 out of their 1 million Android user's devices 90% of them were running out of date versions that have known vulnerabilities. The importance of fixing these vulnerabilities was demonstrated by Grant Hernandez [24], at University of Florida when he showed a proof of concept where he gained root privileges on his Pixel 2 with the CVE-2019-2215 exploit.

Hence, while creating the proof of concept application we presume that the malware is capable of escalating its privileges by exploiting the vulnerabilities in the Android OS. The rest of the thesis assumes that the malware has already gained root privileges.

4. Proof of Concept Android Malware

To create the proof of concept application, first the testing and developing environment needs to be set up. Afterwards the possibilities of how the proof of concept application can achieve its goals of capturing and auto-entering PINs can be explored.

4.1. Development and Test Environment

For developing and testing the proof of concept application an Android emulator was used, which simulates an Android device on a PC with software. The emulator we used came with Android Studio, the official IDE for developing Android software [25].

For creating an emulated device in Android Studio there is a built-in tool called Android Virtual Device Manager. There the user can create an emulator with any Android version, with and without the Google Play Store, and select from a variety of Google devices: Pixel 2, Nexus 5 etc. For developing the proof of concept app Android 7.1.1 without Google Play Store and Pixel 2 XL was selected.

Additionally, the Android running in the emulator needed to be rooted. The normal rooting process for a mobile device usually requires unlocking the bootloader, which the emulator does not have. The process of rooting an emulator is different. It is done using the Android Debug Bridge (abd), which is a command line tool to communicate with the Android device [26]. By running the `root` command in the Android Debug Bridge, the root privileges are given to the shell. Unfortunately, we discovered that the last Android version in the emulator where that command is supported is Android 7 and 7.1.1. Because of this the proof of concept application was developed on Android 7.1.1. Additionally, we discovered that the `root` command is not supported if the Android emulator has Google Play Store present.

```
1:  C:\Users\User\AppData\Local\Android\Sdk\emulator\emulator.exe -avd
emulator1 -writable-system
2:  C:\Users\User\AppData\Local\Android\Sdk\platform-tools\adb.exe root
3:  C:\Users\User\AppData\Local\Android\Sdk\platform-tools\adb.exe remount
4:  C:\Users\User\AppData\Local\Android\Sdk\platform-tools\adb.exe push
C:\Users\User\Downloads\root\x86\su.pie /system/bin/su
```

```
5:  C:\Users\User\AppData\Local\Android\Sdk\platform-tools\adb.exe -e shell
6:  su root
7:  cd /system/bin
8:  chmod 06755 su
9:  su --install
10: su --daemon&
11: setenforce 0
```

Figure 2. Commands executed to obtain root access to the emulator [27].

We followed instructions on StackOverFlow [27] on how to root the emulator. The commands needed to root the emulator are shown in Figure 2, all of which are written into a terminal. Line 1 starts the emulator named `emulator1` with the system image being writable. It is important to start the emulator using this command instead of starting it from the Android Studio GUI, especially after it is rooted, or else the emulator will get stuck booting. The commands from line 2 are written into a second terminal, because terminal 1 is running the emulator. Then in line 2 using the Android Debug Bridge (adb.exe) we enable root permissions in the adb shell, so when we open the shell of the emulated device we have root permissions. Line 3 remounts the drive. This is needed for the `push` command on the following line. If this is not done, then moving files to the system folder will not work. Line 4 pushes the custom SuperSu `su` file (version 2.44) [28] into the Android file system. The `su` file can later be run by any application in the device to get root permissions. Line 5 opens the adb shell of the emulated device where the rest of the commands are entered. Line 7 navigates to the folder where `su` is and line 8 changes the file permission so everyone can read and execute it. Line 6, 9 and line 10 set up `su` for use. Line 11 turns off SELinux protection and sets it to permissive to remove its protections. Otherwise running `/system/bin/su` in the shell will not give the shell root access.

After following these steps the emulator is set up and rooted. Now every app can run `/system/bin/su` which will give the shell or Java `Process` class where it was run root permissions.

4.2. Implementation of Malware Activities

The goal of the proof of concept application is to capture the PINs the user inputs in the Smart-ID app and later to enter the captured PINs automatically.

To achieve this goal, the proof of concept application had to achieve 4 distinct tasks. First, the PoC needed to detect the Smart-ID transaction screen, to know when to run its malicious code. Secondly, the PoC needed to get information out of the Smart-ID transaction screen, for example, the locations of the Smart-ID PIN numpad buttons, to know if the user touch correlates with a Smart-ID button. Additionally, text, whether Smart-ID requests PIN1 or PIN2 would be useful, although depending on the approach, it might not be needed. Furthermore, the proof of concept application needed to capture the user interactions in the Smart-ID app, to know which of the PIN buttons the user touched. Lastly, the proof of concept application required a way to automatically enter the PIN codes in the Smart-ID transaction screen.

It is important to mention, that all the Android commands are executed by the proof of concept application by using the Java `Process` class. First the `/system/bin/su` is run to get root access in the `Process` and later the commands listed in the following sections.

4.2.1. Detecting the Smart-ID Transaction Window

The proof of concept application has to run its malicious code only when the Smart-ID transaction view is present. For this reason, the PoC needs to be able to check the name of the app that is in the foreground.

To achieve this the proof of concept application uses the `dumpsys window windows | grep "mCurrentFocus"` command to find what app is currently in the foreground. As the name suggests the `dumpsys` command outputs a heap of information about the system and the `window windows` parameters are there to reduce the amount of data this command outputs for efficiency, since the PoC app is going to run this command often. The pipe symbol ("|") pipes the output to the `grep "mCurrentFocus"` command, which finds and outputs the lines that contain the string `mCurrentFocus`. The value of `mCurrentFocus` in the `dumpsys` output shows the name and class of the application in

the foreground. In the case of Smart-ID the value of `mCurrentFocus` contains the string `com.smart_id/com.stagnationlab.sk.TransactionActivity` when the Smart-ID transaction window is in front, as shown in Figure 3.

```
mFocusedActivity: ActivityRecord{452c209 u0
com.smart_id/com.stagnationlab.sk.TransactionActivity
 t20}
```

Figure 3. result of the command `dumpsys window windows | grep "mCurrentFocus"`.

By running the command `dumpsys window windows | grep "mCurrentFocus"` in a loop the PoC application is able to constantly monitor if the Smart-ID transaction view is in the foreground by the presence of `com.smart_id/com.stagnationlab.sk.TransactionActivity` in the output.

4.2.2. Locating Buttons and Reading Text

When the proof of concept application detects the Smart-ID transaction view, then it needs to extract information out of the Smart-ID view. The PoC app needs to know where the PIN numpad buttons (the backspace button, the cancel transaction button and the buttons for numbers 0-9) are on the screen to know if the coordinates of an event it captured point to a Smart-ID PIN button and to know where to send touch events when automatically entering a PIN. Optionally it would be nice if it was possible to get text off the screen, to know whether Smart-ID requests PIN1 or PIN2 or if the PIN entered was correct or incorrect, but depending on the approach this information might not be necessary.

One approach to achieve this would be to use the Smart-ID “activity_transaction.xml” file, that is located in the \resources\res\layout folder in the Smart-ID APK file. This file holds the layout of the Smart-ID transaction view in XML format. From this file it is possible to get the margins, paddings etc. of elements and use them to dynamically calculate the locations of every element, including the PIN numpad buttons. Unfortunately, this approach will not yield the text, whether Smart-ID requests PIN1 or PIN2.

Another way is to capture the layout of the view in the foreground into a XML file. That XML file will show the elements with their names, values, settings and locations. Afterwards that XML file can be read by the malicious app to get the coordinates of the PIN numpad buttons and whether Smart-ID requests PIN1 or PIN2. Android has a tool `uiautomator dump` that does just this. Unfortunately running `uiautomator dump` when the Smart-ID transaction view is in front outputs “ERROR: could not get idle state”. This is a common error with this tool in some apps, since `uiautomator` requires the app in the foreground to be idle for 1000 ms [29]. Fortunately, there are third party tools available that build on the `uiautomator` command, for example, appium-uiautomator2 [30], that could be used to potentially get around this problem.

The third way to get information out of the foreground app’s view is to use the `dumpsys activity top` command. This command outputs a slew of information about the app in the foreground, including the hierarchy of the view in front in XML format. This hierarchy shows the names of the elements, their margins (`mLeft`, `mRight`, `mTop`, `mBottom`) in their layouts and the hierarchy of the elements. Unfortunately it does not show what text a text element is displaying or if an element is visible or not. This means that this method would yield the locations of the PIN buttons in Smart-ID, but not the text, whether the app requests PIN1 or PIN2.

The most complex approach to achieve the goal of this section would be to use image recognition. Specifically, OpenCV for Android [31]. OpenCV is a library designed for real-time computer vision. The idea of this approach is that with the command `screencap /mnt/sdcard/Download/screenshot.png` the app takes a screenshot, where OpenCV can use template matching to find the Smart-ID PIN numpad button locations, since the coordinates on the screenshot are going to be the coordinates on the screen in the real Smart-ID app. Template matching is a method where the software tries to locate a smaller image in a larger image. The larger image in this case is the screenshot and the smaller image is the image of the Smart-ID PIN button that we made, like the one shown in Figure 4. And after finding a match, the app would know the location of the Smart-ID PIN button.

Figure 4. Image of button 1 used for template matching.

Furthermore, OpenCV can be used to perform OCR (Optical Character Recognition) to read the text off the screenshot. Using this feature, it is possible to read off the screenshot which PIN Smart-ID requests. This would tell the malware which PIN the user entered and later which PIN the malware needs to automatically enter.

Using OpenCV has two downsides. Firstly, image recognition is not fast, meaning the use of template matching needs to be strategic, using it only when necessary, for example, when the Smart-ID first appears in the foreground. Secondly, Smart-ID does not permit taking screenshots by default, although the user can disable this option in the settings. This limitation is even there when the malware has root privileges. This means the malware would need to bypass this protection.

Since the proof of concept app is for demonstration purposes only, it uses hardcoded coordinate adjusted for the Pixel 2 XL smartphone.

4.2.3. Capturing PIN Codes

When the proof of concept application knows that the Smart-ID transaction view is open and has found the locations of the Smart-ID PIN numpad buttons, then it can start capturing the PINs. To capture a PIN, the proof of concept application captures touch events in the Smart-ID transaction screen, extracts the coordinates from those events and checks if the captured event coordinates are on any of the Smart-ID PIN button locations.

There are multiple ways to capture touch events in the Android OS, but all require access to the `/dev/input/event1` file, which is a node for touch events in the OS. The input events that take place in the device can be seen with the `getevent` command when the

command is running and the device screen is touched as shown in Figure 5. The `getevent` command shows information about the input events and provides a live dump of kernel input events [32]. Using the `getevent` command with the parameter `-l`, the output will use the label names, that hint what each line means.

```
generic_x86:/ # getevent
/dev/input/event1: 0003 0039 00000000
/dev/input/event1: 0003 0030 00000003
/dev/input/event1: 0003 003a 00000081
/dev/input/event1: 0003 0035 00002a38
/dev/input/event1: 0003 0036 00005d49
/dev/input/event1: 0000 0000 00000000
/dev/input/event1: 0003 003a 00000000
/dev/input/event1: 0003 0039 ffffffff
/dev/input/event1: 0000 0000 00000000

generic_x86:/ # getevent -l
/dev/input/event1: EV_ABS       ABS_MT_TRACKING_ID   00000000
/dev/input/event1: EV_ABS       ABS_MT_TOUCH_MAJOR   00000003
/dev/input/event1: EV_ABS       ABS_MT_PRESSURE      00000081
/dev/input/event1: EV_ABS       ABS_MT_POSITION_X    00002a38
/dev/input/event1: EV_ABS       ABS_MT_POSITION_Y    00005d49
/dev/input/event1: EV_SYN       SYN_REPORT           00000000
/dev/input/event1: EV_ABS       ABS_MT_PRESSURE      00000000
/dev/input/event1: EV_ABS       ABS_MT_TRACKING_ID   ffffffff
/dev/input/event1: EV_SYN       SYN_REPORT           00000000
```

Figure 5. Output of the `getevent` command when the display is touched.

Unfortunately, the proof of concept application cannot use the `getevent` command to capture events. When used with the Java `Process` class, which the PoC app uses to execute commands and to read a command's output, the `getevent` command has a buffer that prevents the command from outputting the events instantly.

To get around this, the proof of concept application uses the `od /dev/input/event1` command to capture the touch events in the OS. The `od` command converts input into octal format [33]. This command is a convenient alternative to the `getevent` command, because its output is easily processed by the proof of concept application, as opposed to using `cat /dev/input/event1` where the output is binary. Furthermore, the `od` command has no output buffer.

```
generic_x86:/ # od /dev/input/event1
0000000  132552  057060  063057  000010  000003  000071  000000  000000
0000020  132552  057060  063057  000010  000003  000060  000310  000000
```

```
0000040  132552  057060  063057  000010  000003  000072  000201  000000
0000060  132552  057060  063057  000010  000003  000065  036265  000000
0000100  132552  057060  063057  000010  000003  000066  044540  000000
0000120  132552  057060  063057  000010  000000  000000  000000  000000
0000140  132552  057060  177762  000010  000003  000072  000000  000000
0000160  132552  057060  177762  000010  000003  000071  177777  177777
0000200  132552  057060  177762  000010  000000  000000  000000  000000
```

Figure 6. Output of the `od` command when the display is touched.

After capturing a touch event with the `od /dev/input/event1` command the proof of concept application extracts the coordinates from the event. It does this by searching for the keys `000065` (`ABS_MT_POSITION_X`) and `000066` (`ABS_MT_POSITION_Y`) in the seventh column. The correct key values and locations were found by comparing the output shown in Figure 5 with the output shown in Figure 6. After finding a match the PoC takes the value from the next column and after converting the value to decimal, it has the event coordinate.

This is the process the proof of concept app goes through when it is just a simple touch event, but there are more types of events in Android and each one has a different structure. Firstly, when the user taps the screen in the same place multiple times, then the OS excludes the coordinates from the events after the first one. In this case the proof of concept application finds the event coordinates by copying the previous event coordinates it captured. The proof of concept app handles dragging events similarly. A dragging event has multiple sets of coordinates, sometimes missing either x or y in them, depending on the movement. Although dragging events have no special functionality in the Smart-ID app, the user still might slightly move their finger when pressing the PIN button and this is still recognized as a button press by the Smart-ID app. The final event type is the multi touch events like pinching to zoom. These events are even more complex, but they can be recognized by the `ABS_MT_SLOT` tag or `000057` in the `od /dev/input/event1` command output. The multi touch events are ignored by the PoC application, because they have no functionality in the Smart-ID app.

It is important to note that all the coordinates captured by the proof of concept application are absolute coordinate not screen coordinates. The difference is that screen coordinates scale is the resolution, while absolute coordinates are in a different scale. For example,

Pixel 2 XL that we tested it on has absolute coordinates scale 32767x32767 and screen coordinates in scale 2880x1440 (the resolution). But the proof of concept application needs the event coordinates in screen coordinates, because the Smart-ID button locations are captured in the screen coordinates scale.

To convert from one coordinate system to another the proof of concept application uses a simple equation.

```
multiplier = screen-max / absolute-max
```

In this equation the `screen-max` is the maximum screen coordinates in one axis. To get the maximum screen coordinates the proof of concept application uses the `dumpsys window displays` command, where the maximum screen coordinates (the device resolution) are listed under `init`. The `absolute-max` is the maximum absolute coordinate in one axis. The proof of concept application uses the command `getevent -li`, where the maximum absolute coordinates values are listed under `ABS_MT_POSITION_X` and `ABS_MT_POSITION_Y`.

```
screen-coordinate = multiplier * absolute-coordinate
```

The resulting `multiplier` is multiplied with an absolute coordinate to get a corresponding screen coordinate. This `multiplier` is created for both the x and y axis.

After capturing a PIN entered by the user in the Smart-ID transaction screen the malware needs to know which PIN it was. If, for example, the OpenCV method explained in the previous section was used, then the malware could read which PIN the Smart-ID requested and the malware would know which PIN it captured. But since the proof of concept application does not read information off the Smart-ID transaction screen, it cannot be certain which PIN it captured. To get around this issue the proof of concept application uses a different strategy. The proof of concept assumes that the PINs have the default lengths: PIN1 is 4 digits long and PIN2 is 5 digits long. This means that if the proof of concept application captures a PIN that is 4 digits long, then it is saved as PIN1 and if the PIN is 5 digits long, then it is saved as PIN2. The downside of this approach is

that the proof of concept cannot handle a situation where the PINs do not have the default lengths. Although not implemented a possible fix would be to save the shorter captured PIN as PIN1 and the longer captured PIN as PIN2.

4.2.4. Entering PIN Codes

When the proof of concept application has learned the PINs, it can then automatically enter them in the Smart-ID transaction window. To enter a PIN, the PoC application sends touch events to the corresponding Smart-ID PIN numpad button locations. For example, if the proof of concept application has captured a PIN with the value of 2671, then the PoC sends touch events to the Smart-ID button 2, 6, 7 and 1 locations, in that order.

To send touch events in Android, one method would be to use the `sendevent` command. The `sendevent` command works by essentially sending every line captured with the `getevent` command back with the `sendevent` command, with all the values converted from hexadecimal to decimal. The downside of using `sendevent` to send touch events is that, as seen with the `getevent` command (Figure 5) a simple touch event has 9 lines and to send this event back, there needs to be matching 9 lines of `sendevent` commands. Because of the added complexity the `sendevent` command brings, the `input tap` command was used instead.

To send an event the proof of concept application uses the `input tap x y` command [34]. This command sends a touch event to the x and y coordinates on the screen. The x and y coordinates here are screen coordinates, as opposed to absolute coordinates, which is more convenient because the PoC application already has the Smart-ID PIN numpad button coordinates saved as screen coordinates.

Using the `input tap x y` command, the proof of concept app enters captured PINs. First the PoC application enters PIN1. This is done because PIN1 (4 digits) is by default shorter than PIN2 (5 digits) and this would not trigger the invalid PIN error. When the Smart-ID transaction screen does not close after entering the PIN the proof of concept application assumes that Smart-ID expects PIN2. Then the proof of concept application clears the entered PIN by sending touch events to the backspace button and afterwards

enters PIN2. If the Smart-ID app is still open, then there is some fault and the proof of concept application presses the Cancel button to end the transaction. Due to this strategy even if the PINs are mixed up, it would still be able to complete the transaction, although the invalid PIN error in Smart-ID would be triggered.

4.3. Proof of Concept Application

The proof of concept application was developed on an emulated Pixel 2 XL running Android 7.1.1. and tested with the Smart-ID app version 18.3.175. The source code for the proof of concept application is available on Github [35]. The APK file that can be installed on an Android device is also on Github [36]. The videos demonstrating the app working are available on Google Drive [37].

Figure 7. Front end of the proof of concept malware application. The malware is running, auto-entering of PINs is disabled and the PIN1 has been learned by the app.

The proof of concept application has a simple UI as shown in Figure 7. There is a button to start and another button to stop the malicious code and an additional switch to activate the auto-entering of PINs. The text above the buttons lets the user know if the malware is running or not and the captured PINs are displayed below the buttons.

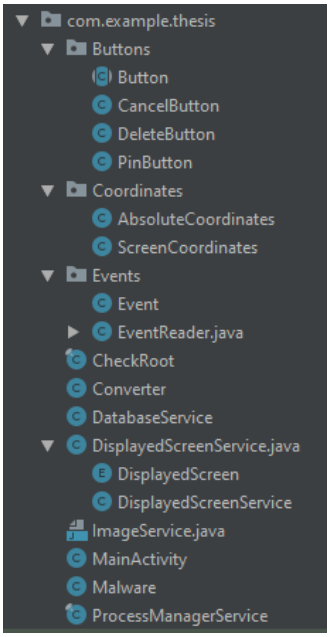

Figure 8. Proof of concept application classes.

4.3.1. Capturing PIN Codes

When the user presses the Start Malware button in the UI, then the malicious part of the application starts, with the task of capturing the PINs entered in the Smart-ID transaction screen. The process of capturing a PIN in the proof of concept app is as follows:

1. The proof of concept application is opened and the `MainActivity` class is run. The `MainActivity` class holds the logic for the UI actions like button presses and also specifies the layout XML files that details the application's UI.
2. The Start Malware button is pressed, which starts a `Malware` class thread.
3. The Malware thread runs a loop where the `DisplayedScreenService` checks every second if the Smart-ID transaction screen is in front.
4. The `DisplayedScreenService` detects the Smart-ID transaction screen.
5. The `DisplayedScreenService` locates the Smart-ID PIN buttons in the transaction screen and creates corresponding `Button` class objects for each of the Smart-ID PIN numpad buttons. The `Button` objects detail the height, width, value of the button and the button's top left corner `ScreenCoordinate`. Currently the `Button` objects are created with hardcoded values for Pixel 2 XL.

6. If auto-entering PINs is not enabled, then the `Malware` thread starts the `EventReader` class thread and continues to check in a loop if the Smart-ID transaction screen is in the foreground.
7. The created `EventReader` thread captures touch events in the Smart-ID transaction screen.
8. When a touch event is captured the `EventReader` checks which type of event it is (simple touch event, dragging event, multiple touches to the same place or multi touch event) by examining the pattern of the event. The event type is needed to reliably extract the coordinates from the events, due to the disparity in different event type outputs.
9. The `EventReader` extracts the coordinates from the captured event and creates a corresponding `Event` object, that holds the events `AbsoluteCoordinates`.
10. When the `Malware` thread discovers that the Smart-ID transaction screen is no longer present, then the `EventReader` thread is stopped.
11. The `Malware` class sends the captured `Event` objects to the `DisplayedScreenService`.
12. The `DisplayedScreenService` checks if the event coordinates are located on any of the `Button` objects created earlier. If the event coordinates are located on a PIN number button, then the value of the button is saved, if the coordinates are on the backspace button, then the previous saved value is deleted and if the coordinates are on the Cancel button, then everything is discarded.
13. If the Cancel button was not pressed, then the proof of concept application assumes the transaction was successful.
14. The `DisplayedScreenService` returns the captured PIN to the `Malware` thread.
15. The `Malware` thread sends the PIN to the `DatabaseService` that saves the PIN into the SQLite database. If the captured PIN has 4 digits, then it is saved as PIN1, if the captured PIN was 5 digits, then it is saved as PIN2.
16. The captured PIN is displayed in the proof of concept application's UI.

4.3.2. Entering PIN Codes

After learning a Smart-ID PIN, the proof of concept application can start entering the PIN automatically. The process of automatically entering PINs in the Smart-ID transaction screen is as follows:

1. The first 5 steps described in Section 3.3.1 are done, with the addition of the “Auto enter PIN” switch needing to be enabled.
2. If auto-entering PINs is enabled, then the `Malware` thread sends PIN1 to the `DisplayedScreenService`.
3. The `DisplayedScreenService` enters PIN1 by sending touch events to the locations of the Smart-ID PIN buttons.
4. The `Malware` thread waits 4 seconds to give time for the Smart-ID transaction screen to close.
5. If the Smart-ID transaction screen is closed, then the proof of concept application concludes that PIN1 was asked and was accepted as valid. If the Smart-ID transaction screen is still open, then the wrong PIN was entered.
6. If the Smart-ID transaction screen did not close, then the `Malware` thread tells the `DisplayedScreenService` to remove the entered PIN1 digits.
7. The `DisplayedScreenService` presses the backspace button in the Smart-ID transaction screen to clear out PIN1. This is done because by default PIN1 is shorter than PIN2 and to enter PIN2 next, the previously entered PIN1 digits need to be removed.
8. PIN2 is entered.
9. The `Malware` thread waits 4 seconds to give time for the Smart-ID transaction screen to close.
10. If the Smart-ID transaction screen is closed, then the proof of concept application concludes that PIN2 was asked and was accepted as valid. If the Smart-ID transaction screen is still open, then the proof of concept application assumes something is wrong and presses the Cancel button to end the transaction.

4.3.3. Requirements for Real World Use

The proof of concept application can capture and enter PINs in the Smart-ID application, but for a malware to be used in the real world some extra functionality would be needed. We believe that all this functionality can be implemented with additional effort.

First and foremost, the malware would need to get into the device and obtain root privileges. This would most likely mean getting the malicious application on the Google Play Store, disguised as a regular application, but would contain known root exploits to gain root privileges on Android devices. Once on the device the malicious application would need to exploit some vulnerability in Android to get root access or at least access to the needed files and commands. In case the device was not originally rooted the fact that the device is rooted should be hid from the Smart-ID app, to prevent the user from becoming suspicious of malicious behavior.

Secondly, the malicious application would need to be able to communicate with the attacker, in order for the attacker to let the malware know which transactions need the PINs automatically entered. The attacker would also need to be able to send the malware the correct Verification Code in the case where the correct Verification Code needs to be selected among 3.

Moreover, the malicious app should start automatically during boot and should work secretly in the background without the user's knowledge. In addition, the malware should not allow the user to delete nor close it.

Additionally, the malware has to work on different devices with different screen resolutions and proper button locating needs to be implemented. Any of the methods described in Section 3.2.2 would work. The easiest would be to use the `dumpsys activity top` command.

5. Conclusion

This thesis explored the possibilities of how a malicious application with root privileges can interact with the Smart-ID application on Android with the intent of capturing the user's PINs and later automatically entering the PINs into the Smart-ID transaction screen. The proof of concept application demonstrated that such actions are possible and while the proof of concept application achieved the goals set in this thesis, it still lacks some functionality that a real malware would need to be used in the real world. We believe that with additional effort these features could be implemented, but the purpose of this work was to only show that the scenario where malicious code can interact with the Smart-ID application is possible.

Despite the many security features Smart-ID has, there is little that can be done for security when the malware has root permissions and even though no vulnerabilities were found in the Smart-ID application, in such a case it did not matter, since the malware was still able to achieve its goal. The findings of this thesis show the importance to Android Smart-ID users of making sure that their device's Android OS is up-to-date and to be careful when installing third-party applications on their devices.

6. References

[1] Smart-ID services. https://www.smart-id.com/services/ (03.12.2019)

[2] Smart-ID. https://www.smart-id.com/ (29.12.2019)

[3] Nüüd saab ametlikku digiallkirja anda ka Smart-ID-ga, Delfi Forte, 05.02.2020. https://forte.delfi.ee/news/digi/tanasest-saab-ametlikku-digiallkirja-anda-ka-smart-id-ga?id=88854745 (07.05.2020)

[4] Buldas A., Kalu A., Laud P. and Oruaas M. Server-Supported RSA Signatures for Mobile Devices, 2017. https://research.cyber.ee/~peeter/research/esorics2017.pdf (04.01.2020)

[5] Smart-ID authentication process image. https://www.smart-id.com/wordpress/wp-content/uploads/2017/05/EN-SID-on-iOS.png (30.12.2019)

[6] Smart-ID technical documentation. https://github.com/SK-EID/smart-id-documentation/wiki/Technical-overview (04.01.2020)

[7] FLAG_SECURE, Android Documentation. https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_SECURE (04.01.2020)

[8] Smart-ID App Update Required, Smart-ID FAQ. https://www.smart-id.com/help/faq/warnings-and-limitations/smart-id-update-required/ (31.03.2020)

[9] Smart-ID Warnings and Limitations: Outdated Operating System, Smart-ID FAQ. https://www.smart-id.com/help/faq/warnings-and-limitations/outdated-operating-system (09.01.2020)

[10] Smart-ID Warnings and Limitations: Using Smart-ID with a Rooted Device, Smart-ID FAQ. https://www.smart-id.com/help/faq/warnings-and-limitations/using-smart-id-with-a-rooted-device (09.01.2020)

[11] Smart-ID Security. https://www.smart-id.com/security/ (03.01.2020)

[12] Why do I sometimes see one confirmation code and sometimes three, Smart-ID FAQ. https://www.smart-id.com/help/faq/using-smart-id/why-do-i-sometimes-see-one-confirmation-code-and-sometimes-three/ (28.04.2020)

[13] Operating System Market Share Worldwide, GlobalStats Statcounter, 11.2019. https://gs.statcounter.com/os-market-share/ (03.12.2019)

[14] What is Android. https://www.android.com/what-is-android/ (03.12.2019)

[15] System and kernel security, Android Open Source Project. https://source.android.com/security/overview/kernel-security (03.12.2019)

[16] Application Sandbox, Android Open Source Project. https://source.android.com/security/app-sandbox (03.12.2019)

[17] Content providers, Android Documentation. https://developer.android.com/guide/topics/providers/content-providers.html (28.04.2020)

[18] Permission Overview, Android Documentation. https://developer.android.com/guide/topics/permissions/overview (28.04.2020)

[19] Vulnerability Alerts, TheBestVpn, 06.03.2020. https://thebestvpn.com/vulnerability-alerts/ (28.04.2020)

[20] Samsung One UI. https://www.samsung.com/global/galaxy/apps/one-ui/ (30.12.2019)

[21] OnePlus OxygenOS. https://www.oneplus.com/ee/oxygenos (30.12.2019)

[22] Nohl K. and Lell J. The Android ecosystem contains a hidden patch gap, Security Research Labs, 2018. https://srlabs.de/bites/android_patch_gap/ (31.03.2020)

[23] Duo Security Finds Over 90 Percent of Android Devices Run Outdated Operating Systems, Duo Security Press release, 19.01.2016. https://duo.com/about/press/releases/duo-security-finds-over-90-percent-of-android-devices-run-outdated-operating-systems (30.12.2019)

[24] Hernandez G. Tailoring CVE-2019-2215 to Achieve Root, 15.10.2019. https://hernan.de/blog/2019/10/15/tailoring-cve-2019-2215-to-achieve-root/ (30.12.2019)

[25] Android Studio. https://developer.android.com/studio (31.03.2020)

[26] Android Debug Bridge (adb), Android Developers Documentation. https://developer.android.com/studio/command-line/adb (31.03.2020)

[27] How to get root access on Android emulator, StackOverFlow, 01.06.2017. https://stackoverflow.com/questions/5095234/how-to-get-root-access-on-android-emulator (31.03.2020)

[28] SuperSU files for Android. https://androidfilehost.com/?w=files&flid=154643 (31.03.2020)

[29] Source code for the `uiautomator dump` command. https://android.googlesource.com/platform/frameworks/testing/+/jb-mr2-release/uiautomator/cmds/uiautomator/src/com/android/commands/uiautomator/DumpCommand.java (29.04.2020)

[30] Appium's uiautomator2-server repository. https://github.com/appium/appium-uiautomator2-server (29.04.2020)

[31] OpenCV downloads page. https://opencv.org/releases/ (31.03.2020)

[32] The `getevent` tool, Android Open Source Project. https://source.android.com/devices/input/getevent (31.03.2020)

[33] The `od` command, Linux Documentation. http://man7.org/linux/man-pages/man1/od.1.html (31.03.2020)

[34] Input.java, Android in Google Git. https://android.googlesource.com/platform/frameworks/base/+/HEAD/cmds/input/src/com/android/commands/input/Input.java (31.03.2020)

[35] The proof of concept application's source code's repository. https://github.com/silvergithub999/Thesis (19.04.2020)

[36] The Proof of Concept Application APK file. https://github.com/silvergithub999/Thesis/releases (05.05.2020)

[37] The Proof of Concept Application Video Demos. https://drive.google.com/open?id=1YEuXRH0p6KgSy1v5NmRt9uPYcuKVSIjQ (19.04.2020)

7. Appendix

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Silver Maala

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,

A Proof of Concept Malware for Interacting with the Smart-ID Android Application,

supervised by Arnis Paršovs.

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.

Silver Maala
08/05/2020