

Disassembling the APK

As we discussed in the previous section, the Android system uses APKs to distribute and install applications. APK files are archived files containing the source code, assets, and resource files needed for the app to run. In the following paragraphs, we will analyze the content of an APK file by using tools to decompile the app's source code to a more human-readable language, then decode the assets and resources to read any configuration files.

To perform this analysis, we will use [APKTool](#) — a powerful reverse engineering utility that allows us to decode the APK's resources and disassemble the compiled files, enabling inspection and even modification. APKTool also supports rebuilding the APK after changes have been made.

Let's begin by downloading and installing APKTool. It's important to use the latest version to ensure compatibility with apps built using the newest Android Studio versions. The latest release be found on the official [Install Guide](#) for APKTool. Under the [Linux](#) section, we can find the installation steps.

Linux

- 1. Download the Linux [wrapper script](#). (Right click, Save Link As `apktool`)
- 2. Download the [latest version](#) of Apktool.
- 3. Rename the downloaded jar to `apktool.jar`.
- 4. Move both `apktool.jar` and `apktool` to `/usr/local/bin`. (root needed)
- 5. Make sure both files are executable. (`chmod +x`)
- 6. Try running `apktool` via CLI.

Start by right-clicking the [wrapper script](#) link and selecting [Copy Link Address](#), then run the following command to download it:

Disassembling the APK

```
r11k@htb[/htb]$ wget https://raw.githubusercontent.com/iBotPeaches/Apktool/master/scripts/linux/apktool
```

Next, click on the [latest version](#) link.

Apktool

<>

Source

🔗

Commits

🌿

Branches

🔗

Pull requests

🔄

Pipelines

☁️

Deployments

🔧

Jira issues

🛡️

Security

📄

Downloads

Connor Tumbleson / Untitled project / Apktool

Downloads

📘

For large uploads, we recommend using the API. [Get instructions](#)

Downloads

TagsBranches

Name	Size	Uploaded by	Downloads	Date
Download repository	301.9 MB			
apktool_2.11.1.jar	23.4 MB	Connor Tumbleson	52872	2025-03-11
apktool_2.11.0.jar	23.3 MB	Connor Tumbleson	102819	2025-01-15
apktool_2.10.0.jar	23.2 MB	Connor Tumbleson	208701	2024-09-17

Right-click on the latest JAR version (in this example: [apktool_2.11.1.jar](#)) and select [Copy Link Address](#). Then, issue the following commands to complete the installation.

Disassembling the APK

```
r11k@htb[/htb]$ wget https://bitbucket.org/iBotPeaches/apktool/downloads/apktool_2.11.1.jar
r11k@htb[/htb]$ mv apktool_2.11.1.jar apktool.jar
r11k@htb[/htb]$ sudo mv apktool /usr/local/bin
r11k@htb[/htb]$
```

```
r11k@htb[/htb]$ sudo mv apktool.jar /usr/local/bin
r11k@htb[/htb]$ sudo chmod +x /usr/local/bin/apktool
r11k@htb[/htb]$ sudo chmod +x /usr/local/bin/apktool.jar
```

Once installed, we can use APKTool to disassemble an APK and decode its resources:

Disassembling the APK

```
r11k@htb[/htb]$ apktool d myapp.apk
r11k@htb[/htb]$ ls -l myapp

total 16
-rw-r--r--  1 bertolis  bertolis  1779 Jun 15 13:21 AndroidManifest.xml
drwxr-xr-x  3 bertolis  bertolis   96 Jun 15 13:21 META-INF
-rw-r--r--  1 bertolis  bertolis  2759 Jun 15 13:21 apktool.yml
drwxr-xr-x  3 bertolis  bertolis   96 Jun 15 13:21 assets
drwxr-xr-x  9 bertolis  bertolis  288 Jun 15 13:21 kotlin
drwxr-xr-x  6 bertolis  bertolis  192 Jun 15 13:21 lib
drwxr-xr-x  4 bertolis  bertolis  128 Jun 15 13:21 original
drwxr-xr-x 150 bertolis  bertolis 4800 Jun 15 13:21 res
drwxr-xr-x  8 bertolis  bertolis  256 Jun 15 13:21 smali
drwxr-xr-x  3 bertolis  bertolis   96 Jun 15 13:21 unknown
```

The **myapp** directory now contains the decoded contents of the APK. One key file of interest is **AndroidManifest.xml**. As we discussed in the Android Fundamentals module, this file is encoded in APKs and cannot be read by simply unzipping the APK. Fortunately, APKTool has decoded it, and we can now examine its contents

Understanding the Manifest

Open the manifest file using a text editor like **vim** or **vscode**:

Disassembling the APK

```
r11k@htb[/htb]$ vim myapp/AndroidManifest.xml
```

Code: xml

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:compileSdkVersion="32" android:compileSdkVersionCodename="12"
  package="com.example.myapp" platformBuildVersionCode="32"
  platformBuildVersionName="12">
  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE"/>
  <application android:allowBackup="true" android:appComponentFactory="androidx.core.app.CoreComponentFactory"
    android:dataExtractionRules="@xml/data_extraction_rules" android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher" android:label="@string/app_name" android:name="com.example.myapp.Init"
    android:networkSecurityConfig="@xml/network_security_config" android:requestLegacyExternalStorage="true"
    android:roundIcon="@mipmap/ic_launcher_round" android:supportRtl="true" android:theme="@style/Theme.MyApp">
    <activity android:exported="true" android:name="com.example.myapp.MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
      <meta-data android:name="android.app.lib_name" android:value=""/>
    </activity>
    <activity android:exported="true" android:name="com.example.myapp.MenuActivity">
      <meta-data android:name="android.app.lib_name" android:value=""/>
      <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
```

```
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:host="myapp" android:scheme="app"/>
    </intent-filter>
</activity>
<provider android:name="com.example.myapp.MyProvider" android:authorities="com.provider"/>
<provider android:authorities="com.example.myapp.androidx-startup" android:exported="false"
    android:name="androidx.startup.InitializationProvider">
    <meta-data android:name="androidx.emoji2.text.EmojiCompatInitializer" android:value="androidx.startup"/>
    <meta-data android:name="androidx.lifecycle.ProcessLifecycleInitializer" android:value="androidx.startup"/>
</provider>
<service
    android:name="com.example.myapp.MyService"
    android:exported="false"
    android:icon="@drawable/ic_launcher" >
</service>
</application>
</manifest>
```

Package Name

The `<manifest>` tag contains several important attributes. One of the first things to look at is the package name, which uniquely identifies the application:

Code: `xml`

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:compileSdkVersion="32" android:compileSdkVersionCodename="12"
    package="com.example.myapp" platformBuildVersionCode="32"
    platformBuildVersionName="12">
```

The package name (`com.example.myapp`) is essential when performing both static and dynamic analysis. From the same snippet, we can see the target SDK version is 32, which is also valuable information as different Android SDK versions may present different security implications.

Permissions

Apps will sometimes request permissions from the user while running, attempting to access various functionalities. The following snippet shows these permissions declared in the manifest.

Code: `xml`

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

Overly permissive apps or those with poor security hygiene may request more permissions than necessary, which can introduce risk. For example, storing sensitive data in external storage may allow other apps to access that data if they also have `Read` permissions. Below is a table summarizing some permissions that a pentester should carefully evaluate:

Permissions	Description
<code>READ_SMS</code> , <code>SEND_SMS</code> , <code>RECEIVE_SMS</code>	Grants the app the ability to read, send, and receive SMS messages. This can be exploited to intercept one-time passwords (OTPs) used in two-factor authentication or to send unauthorized messages.
<code>READ_CALL_LOG</code> , <code>WRITE_CALL_LOG</code>	Allows the app to access and modify the device's call history. This could expose sensitive metadata about the user's communication patterns or be used to conceal malicious calls.
<code>READ_CONTACTS</code> , <code>WRITE_CONTACTS</code>	Provides access to the device's contact list and the ability to alter it. A malicious app could exfiltrate contact data or inject fraudulent entries for social engineering or spam purposes.

ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION	Enables the app to access the device's location. This may allow real-time tracking of the user's physical movements without consent.
READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE	Permits the app to access and modify files stored on external storage. Sensitive information could be exposed or altered, especially if improperly secured by the app or the user.
GET_ACCOUNTS	Lets the app query the list of accounts registered on the device. This information could be misused to target specific accounts or harvest data for phishing attacks.
CAMERA	Provides the app with access to the device's camera hardware. If misused, it could enable covert photo or video capture without the user's knowledge.
RECORD_AUDIO	Grants the app permission to capture audio through the device's microphone. This could be exploited to eavesdrop on conversations in the background.
INSTALL_PACKAGES, REQUEST_INSTALL_PACKAGES	Allows the app to trigger package installations via the system installer. If abused, it could lead to silent installation of additional malicious apps.
SYSTEM_ALERT_WINDOW	Enables the app to draw overlays on top of other apps. This can be leveraged to create deceptive UI elements for phishing or tricking users into performing unintended actions (e.g., clicking "Allow" instead of "Deny").

Application Class

The next element we see is `<application>`, which contains many attributes.

Code: `xml`

```
<application android:allowBackup="true" android:appComponentFactory="androidx.core.app.CoreComponentFactory"
    android:dataExtractionRules="@xml/data_extraction_rules" android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher" android:label="@string/app_name" android:name="com.example.myapp.Init"
    android:networkSecurityConfig="@xml/network_security_config" android:requestLegacyExternalStorage="true"
    android:roundIcon="@mipmap/ic_launcher_round" android:supportsRtl="true" android:theme="@style/Theme.MyApp">
```

An interesting attribute in this manifest is the `android:name="com.example.myapp.Init"`. This attribute is used to specify a custom Application class (subclass of the Application class). The Application class and any subclass, like the `Init` in this example, are instantiated before any other class when the application process is created. This means the `Init` class will run immediately when the app starts and before the user interacts. Developers usually use this for initializations, but penetration testers should pay attention to such classes since, among other reasons that might have a security impact, they are often used for third-party libraries that require initialization. If these libraries have known vulnerabilities or are not securely configured, they could introduce security risks.

Network Security Configurations

Another attribute worth noticing is the `android:networkSecurityConfig="@xml/network_security_config"`. This attribute refers to a file that contains network security configurations of the app, where we can specify custom trusted Certificate Authorities, permit or deny cleartext (HTTP) traffic, and other network-related settings. Let's read the content of this file.

Disassembling the APK

```
r11k@htb[/htb]$ vim myapp/res/xml/network_security_config.xml
```

Code: `xml`

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config cleartextTrafficPermitted="false">
    <domain includeSubdomains="true">192.168.1.20</domain>
    <trust-anchors>
      <certificates src="@raw/certificate" />
    </trust-anchors>
  </domain-config>
</network-security-config>
```

In this example, the attribute `clearTextTrafficPermitted="false"` indicates that only a secure (HTTPS) connection is allowed for communication with the domain `192.168.1.20`. Also, the element `<certificates src="@raw/certificate" />` refers to the custom certificate the app trusts for the secure connection. The certificate file of the app is located in the following directory.

Disassembling the APK

```
r11k@htb[/htb]$ ls -l myapp/res/raw

total 8
-rw-r--r--  1 bertolis  bertolis  1029 Jun 15 14:55 certificate.der
```

The following snippet shows an example of the `network_security_config.xml` file. It implies the implementation of a technique called Certificate Pinning, which adds an extra security layer to the app.

Code: xml

```
<network-security-config>
  <domain-config>
    <domain includeSubdomains="true">example.com</domain>
    <pin-set expiration="2030-12-31">
      <pin digest="SHA-256">7HIpactkIAq2Y49orF00QKurWxmmSFZhBCoQYcRhJ3Y=</pin>
    </pin-set>
  </domain-config>
</network-security-config>
```

Certificate pinning is a security measure that ensures the application is actually connecting to the intended server and not an imposter. Knowing the above configurations, the pen-tester could use the appropriate techniques to bypass this protection layer and intercept the traffic. In both cases, the tester should reconfigure the app to trust another certificate that will be used from the proxy tool to intercept the traffic. Such techniques will be discussed in later sections.

Components

The next snippet of the manifest shows the activities of the application.

Code: xml

```
<activity android:exported="true" android:name="com.example.myapp.MainActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category android:name="android.intent.category.LAUNCHER"/>
  </intent-filter>
  <meta-data android:name="android.app.lib_name" android:value=""/>
</activity>
```

This declares an activity that can be accessed from other components of the app, other apps, or the system. In the above snippet, the `<activity>` element includes the attributes `android:exported="true"` and `android:name="com.example.myapp.MainActivity"`. The `android:name` attribute indicates the name of the Activity, which in case is `MainActivity`. The `android:exported="true"` indicates that the activity is accessible from outside the app, meaning it can be launched by external apps or system components. Because of this, the activity can also be triggered by ADB.

Another important attribute is the `<action android:name="android.intent.action.MAIN"/>` found under the `<intent-filter>` element. This designates the activity as the main entry point of the application, the first screen launched when the user taps the app icon. Only one activity in the app should define this action.

In the snippet below, it shows that the app has two Content Providers and one Service.

Code: xml


```
<provider android:name="com.example.myapplication.MyProvider" android:authorities="com.provider"/>
<provider android:authorities="com.example.myapplication.androidx-startup" android:exported="false"
    android:name="androidx.startup.InitializationProvider">
    <meta-data android:name="androidx.emoji2.text.EmojiCompatInitializer" android:value="androidx.startup"/>
    <meta-data android:name="androidx.lifecycle.ProcessLifecycleInitializer" android:value="androidx.startup"/>
</provider>
<service
    android:name="com.example.myapplication.MyService"
    android:exported="false"
    android:icon="@drawable/ic_launcher" >
</service>
```

All of the component declarations mentioned above are valuable to the pentester, as they provide a clearer understanding of the application. By identifying the application's entry point, the pentester gains a logical starting point for the testing process. Based on the information provided, testers should begin by examining the `com.example.myapplication.MainActivity` class, which serves as the app's entry point, as well as the `com.example.myapplication.Init` class, which extends the `Application` class. As previously noted, this class runs automatically when the app starts, even before `MainActivity`.

The `android:exported="true"` attribute of an `Activity` is critical, as it indicates that the component is accessible to other apps. Among other tests, a pentester must determine whether sensitive information could leak when a malicious application attempts to access this exported Activity.

The following snippet reveals a Deep Link that is used from the `MenuActivity`.

```
Code: xml

<activity android:exported="true" android:name="com.example.myapplication.MenuActivity">
    <meta-data android:name="android.app.lib_name" android:value=""/>
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:host="myapp" android:scheme="app"/>
    </intent-filter>
</activity>
```

The four elements inside the `<intent-filter>` indicate that this Activity uses a Deep Link. Bad programming using Deep Links could lead to security issues and must be thoroughly examined. This is a topic that will be discussed in later sections.

Application Resources

As we look at the files we extracted from the APK file at the beginning of the section using the APKTool, we also see the `lib`, `res`, and `smali` directories. The directory `lib` contains separate shared library (SO) files created for different architectures containing the Native Code of the application. These files can also be retrieved and examined by decompressing the APK file using tools like `unzip`. The directory `res` contains resources like the app's layout, UI strings, images, and other static files. Among other directories, we also notice the directory `values` within the `res` directory. This directory contains XML files with hardcoded strings and integers, such as `Strings.xml`, which stores UI strings as key-value pairs. This file could reveal useful information to a pen-tester, like potential entry points for attack, API keys, secret tokens, database credentials, or cryptographic keys, or help them to understand the application's functionality further. The following is an example of what the `strings.xml` file looks like.


```
Code: xml

<resources>
    <string name="app_name">My App</string>
    <string name="welcome_message">Welcome</string>
    <string name="api_key">1234567890abcdef</string>
    <string name="server_url">http://example.com/api</string>
    <string name="error_invalid_credentials">Invalid username or password</string>
    <string name="error_unauthorized_access">You do not have permission to access this resource</string>
    <string name="password_hint">Your password is the name of your first pet.</string>
```

```
<string name="database_name">user_db</string>
<string name="database_password">db_password123</string>
</resources>
```

In addition, under the `res` directory, we could see the `drawable` and `layout` directories containing images and XML files that define the user interface layouts. Inspecting resources like these can provide useful information about the application's functionality, which can help in better understanding how the application works. Another useful directory that is worth examining during the testing is `xml`. This directory could contain files with network configurations revealing the IP address of potential API endpoints that the app is communicating with. Files like this could also reveal whether the app is told to trust unencrypted connections, an insight that could be taken into consideration in the next steps of the pen-testing process. This directory can also include the sub-directory `raw`, which in turn includes arbitrary files saved in their raw form, like certificates and keys placed there by the developer.

Another interesting directory revealed after extracting the files from the APK using APKTool is the directory `smali`, which contains Smali files. Smali is the human-readable representation of the Dalvik bytecode, which will be the next section's topic.



Connect to Pwnbox

Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location

UK

39ms

Terminate Pwnbox to switch location

Start Instance

/ 1 spawns left

Waiting to start...

Enable step-by-step solutions for all questions

Questions

Answer the question(s) below to complete this Section and earn cubes!

+ 3

What is the application's package name?

Cheat Sheet

Submit your answer here...

+10 Streak pts

Submit

myapp_disass.zip

+ 3

What is the name of the class that is executed immediately after the app launches and before any user interaction occurs?

Submit your answer here...

+10 Streak pts

Submit

+ 3

Which file in Android applications stores UI strings in key-value format?

Submit your answer here...

+10 Streak pts

Submit

← Previous

Next →

- Cheat Sheet
- Go to Questions

Table of Contents

Extracting and Enumerating APK Files

- Introduction
- Disassembling the APK
- Understanding Smali

Analyzing Application's Source Code

- Reading Hardcoded Strings
- Bad Cryptography Implementation
- Reversing Hybrid Apps
- Reading Obfuscated Code
- Deobfuscating Code

Analyzing Native Libraries

- Reversing Shared Objects
- Reversing DLL Files

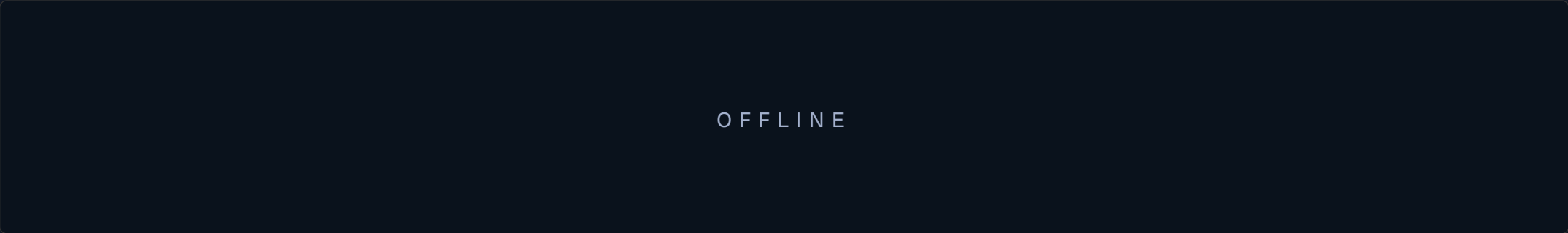
Application Patching

- Authentication Bypass
- Modifying Game Apps
- License Verification Bypass
- Root Detection Bypass

Skills Assessment

- Skills Assessment





O F F L I N E

▶ Start Instance

∞ / 1 spawns left

