# Apktool - Documentation

ibotpeaches.github.io/Apktool/documentation/

## Introduction

### Basic

First lets take a lesson into apk files. Apks are nothing more than a zip file containing resources and assembled java code. If you were to simply unzip an apk like so, you would be left with files such as `classes.dex` and `resources.arsc`.

```
$ unzip testapp.apk
Archive:  testapp.apk
 inflating: AndroidManifest.xml
 inflating: classes.dex
 extracting: res/drawable-
hdpi/ic_launcher.png
 inflating: res/xml/literals.xml
 inflating: res/xml/references.xml
 extracting: resources.arsc
```

However, at this point you have simply inflated compiled sources. If you tried to view `AndroidManifest.xml`. You'd be left viewing this.

```
P4F0\fnversionCodeversionNameandroid*http://schemas.android.com/apk/res/androidpackageplatformBuildVersionCodeplatformBuildVersic
```

Obviously, editing or viewing a compiled file is next to impossible. That is where Apktool comes into play.

```
$ apktool d testapp.apk
I: Using Apktool 2.0.0 on testapp.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with
resources...
I: Loading resource table from file: 1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
$
```

Viewing `AndroidManifest.xml` again results in something much more human readable

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="https://schemas.android.com/apk/res/android" package="brut.apktool.testapp"
platformBuildVersionCode="21" platformBuildVersionName="APKTOOL" />
```

In addition to XMLs, resources such as 9 patch images, layouts, strings and much more are correctly decoded to source form.

### Decoding

The decode option on Apktool can be invoked either from `d` or `decode` like shown below.

```
$ apktool d foo.jar
// decodes foo.jar to foo.jar.out
folder

$ apktool decode foo.jar
// decodes foo.jar to foo.jar.out
folder

$ apktool d bar.apk
// decodes bar.apk to bar folder

$ apktool decode bar.apk
// decodes bar.apk to bar folder

$ apktool d bar.apk -o baz
// decodes bar.apk to baz folder
```

### Building

The build option can be invoked either from `b` or `build` like shown below

```
$ apktool b foo.jar.out
// builds foo.jar.out folder into foo.jar.out/dist/foo.jar file

$ apktool build foo.jar.out
// builds foo.jar.out folder into foo.jar.out/dist/foo.jar file

$ apktool b bar
// builds bar folder into bar/dist/bar.apk file

$ apktool b .
// builds current directory into ./dist

$ apktool b bar -o new_bar.apk
// builds bar folder into new_bar.apk

$ apktool b bar.apk
// WRONG: brut.androlib.AndrolibException: brut.directory.PathNotExist:
apktool.yml
// Must use folder, not apk/jar file
```

> *Info In order to run a rebuilt application. You must resign the application. Android documentation can help with this.*

### Frameworks

Frameworks can be installed either from `if` or `install-framework`, in addition two parameters

- `-p, --frame-path <dir>` - Store framework files into `<dir>`
- `-t, --tag <tag>` - Tag frameworks using `<tag>`

Allow for a finer control over how the files are named and how they are stored.

```
$ apktool if framework-res.apk
I: Framework installed to: 1.apk
// pkgId of framework-res.apk determines number (which is
0x01)

$ apktool if com.htc.resources.apk
I: Framework installed to: 2.apk
// pkgId of com.htc.resources is 0x02

$ apktool if com.htc.resources.apk -t htc
I: Framework installed to: 2-htc.apk
// pkgId-tag.apk

$ apktool if framework-res.apk -p foo/bar
I: Framework installed to: foo/bar/1.apk

$ apktool if framework-res.apk -t baz -p foo/bar
I: Framework installed to: foo/bar/1-baz.apk
```

## Intermediate

### Framework Files

As you probably know, Android apps utilize code and resources that are found on the Android OS itself. These are known as framework resources and Apktool relies on these to properly decode and build apks.

Every Apktool release contains internally the most up to date AOSP framework at the time of the release. This allows you to decode and build most apks without a problem. However, manufacturers add their own framework files in addition to the regular AOSP ones. To use apktool against these manufacturer apks you must first install the manufacturer framework files.

### Example

Lets say you want to decode `HtcContacts.apk` from an HTC device. If you try you will get an error message.

```
$ apktool d HtcContacts.apk
I: Loading resource table...
I: Decoding resources...
I: Loading resource table from file: 1.apk
W: Could not decode attr value, using undecoded value instead: ns=android, name=drawable
W: Could not decode attr value, using undecoded value instead: ns=android, name=icon
Can't find framework resources for package of id: 2. You must install proper framework files, see project website for more
info.
```

We must get HTC framework resources before decoding this apk. We pull `com.htc.resources.apk` from our device and install it

```
$ apktool if
com.htc.resources.apk
I: Framework installed to: 2.apk
```

Now we will try this decode again.

```
$ apktool d HtcContacts.apk
I: Loading resource table...
I: Decoding resources...
I: Loading resource table from file:
/home/brutall/apktool/framework/1.apk
I: Loading resource table from file:
/home/brutall/apktool/framework/2.apk
I: Copying assets and libs...
```

As you can see. Apktool leveraged both `1.apk` and `2.apk` framework files in order to properly decode this application.

### Finding Frameworks

For the most part any apk in `/system/framework` on a device will be a framework file. On some devices they might reside in `/data/system-framework` and even cleverly hidden in `/system/app` or `/system/priv-app`. They are usually named with the naming of "resources", "res" or "framework".

> *Example HTC has a framework called `com.htc.resources.apk`, LG has one called `lge-res.apk`*

After you find a framework file you could pull it via `adb pull /path/to/file` or use a file manager application. After you have the file locally, pay attention to how Apktool installs it. The number that the framework is named during install corresponds to the pkgId of the application. These values should range from 1 to 30. Any APK that installs itself as `127` is `0x7F` which is an internal pkgId.

### Internal Frameworks

Apktool comes with an internal framework like mentioned above. This file is copied to `$HOME/apktool/framework/1.apk` during use.

> *Warning Apktool has no knowledge of what version of framework resides there. It will assume its up to date, so delete the file during Apktool upgrades*

### Managing framework files

Frameworks are stored in different places depending on the OS in question.

- unix - `$HOME/.local/share/apktool`
- windows - `$HOME/AppData/Local/apktool`
- mac - `$HOME/Library/apktool`

If these directories are not available it will default to `java.io.tmpdir` which is usually `/tmp`. This is a volatile directory so it would make sense to take advantage of the parameter `--frame-path` to select an alternative folder for framework files.

Since these locations are in sometimes hidden directories, managing these frameworks becomes a challenge. A simple helper function (added in v2.2.1) allows you to run `apktool empty-framework-dir` to empty out frameworks.

> *Note Apktool has no control over the frameworks once installed, but you are free to manage these files on your own.*

### Tagging framework files

Frameworks are stored in the naming convention of: `<id>-<tag>.apk`. They are identified by pkgId and optionally custom tag. Usually tagging frameworks isn't necessary, but if you work on apps from many different devices and they have incompatible frameworks, you will need some way to easily switch between them. You could tag frameworks by:

```
$ apktool if com.htc.resources.apk -t hero
I: Framework installed to: /home/brutall/apktool/framework/2-hero.apk
$ apktool if com.htc.resources.apk -t desire
I: Framework installed to: /home/brutall/apktool/framework/2-
desire.apk
```

Then:

```
$ apktool d HtcContacts.apk -t hero
I: Loading resource table...
I: Decoding resources...
I: Loading resource table from file: /home/brutall/apktool/framework/1.apk
I: Loading resource table from file: /home/brutall/apktool/framework/2-hero.apk
I: Copying assets and libs...
$ apktool d HtcContacts.apk -t desire
I: Loading resource table...
I: Decoding resources...
I: Loading resource table from file: /home/brutall/apktool/framework/1.apk
I: Loading resource table from file: /home/brutall/apktool/framework/2-
desire.apk
I: Copying assets and libs...
```

You don't have to select a tag when building apk - apktool automatically uses the same tag, as when decoding.

### Smali Debugging

---

> *Warning SmaliDebugging has been marked as deprecated in 2.0.3, and removed in 2.1. Please check SmaliIdea for a debugger.*

### 9Patch Images

Docs exist for the mysterious 9patch images here and there. (Read these first). These docs though are meant for developers and lack information for those who work with already compiled 3rd party applications. There you can find information how to create them, but no information about how they actually work.

I will try and explain it here. The official docs miss one point that 9patch images come in two forms: source & compiled.

- **source** - You know this one. You find it in the source of an application or freely available online. These are images with a black border around them.
- **compiled** - The mysterious form found in apk files. There are no borders and the 9patch data is written into a binary chunk called `npTc`. You can't see or modify it easily, but Android OS can as its quicker to read.

There are problems related to the above two points.

- You can't move 9patch images between both types without a conversion. If you try and unpack 9patch images from an apk and use it in the source of another, you will get errors during build. Also vice versa, you cannot take source 9patch images directly into an apk.
- 9patch binary chunk isn't recognized by modern image processing tools. So modifying the compiled image will more than likely break the `npTc` chunk, thus breaking the image on the device.

The only solution to this problem is to easily convert between these two types. The encoder (which takes source to compiled) is built into the aapt tool and is automatically used during build. This means we only need to build a decoder which has been in apktool since `v1.3.0` and is automatically ran on all 9patch images during decode.

So if you want to modify 9patch images, don't do it directly. Use apktool to decode the application (including the 9patch images) and then modify the images. At that point when you build the application back, the source 9patch images will be compiled.

## Other

### FAQ

**What about the `-j` switch shown from the original YouTube videos?**
Read Issue 199. In short - it doesn't exist.

**Is it possible to run apktool on a device?**
Sadly not. There are some incompatibilities with `SnakeYAML`, `java.nio` and `aapt`

**Where can I download sources of apktool?**
From our Github or Bitbucket project.

**Resulting apk file is much smaller than original! Is there something missing?**
There are a couple of reasons that might cause this.

- Apktool builds unsigned apks. This means an entire directory `META-INF` is missing.
- New aapt binary. Newer versions of apktool contain a newer aapt which optimizes images differently.

These points might have contributed to a smaller than normal apk

**There is no META-INF dir in resulting apk. Is this ok?**
Yes. `META-INF` contains apk signatures. After modifying the apk it is no longer signed. You can use `-c / --copy-original` to retain these signatures. However, using `-c` uses the original `AndroidManifest.xml` file, so changes to it will be lost.

**What do you call "magic apks"?**
For some reason there are apks that are built using modified build tools. These apks don't work on a regular AOSP Android build, but usually are accompanied by a modified system that can read these modified apks. Apktool cannot handle these apks, therefore they are "magic".

**Could I integrate apktool into my own tool? Could I modify apktool sources? Do I have to credit you?**
Actually the Apache License, which apktool uses, answers all these questions. Yes you can redistribute and/or modify apktool without my permission. However, if you do it would be nice to add our contributors (brut.all, iBotPeaches and JesusFreke) into your credits but it's not required.

**Where does apktool store its framework files?**

- **unix** - `$HOME/.local/share/apktool`
- **mac** - `$HOME/Library/apktool`
- **windows** - `$HOME/AppData/Local/apktool`

## Migration Instructions

### v2.2.0 -> v2.2.1

- Update apktool to `v2.2.1`
- `apktool empty-framework-dir --force`

### v2.1.1 -> v2.2.0

- Run the following commands to migrate your framework directory
- Apktool will work fine without running these commands, this will just cleanup abandoned files

- **unix** - `~/.local/share`

  `mkdir -p ~/.local/share; mv ~/apktool`

- **windows** - `%USERPROFILE%\AppData\Local`

  `move %USERPROFILE%\apktool`

### v2.0.1 -> v2.0.2

- Update apktool to `v2.0.2`
- Remove framework file `$HOME/apktool/framework/1.apk` due to internal API update (Android Marshmallow)

### v1.5.x -> v2.0.0

- **Java 1.7 is required**
- Update apktool to v2.0.0
- aapt is now included inside the apktool binary. It's not required to maintain your own aapt install under $PATH. (However, features like `-a / --aapt` are still used and can override the internal aapt)
- The addition of aapt replaces the need for separate aapt download packages. Helper Scripts may be found here
- Remove framework `$HOME/apktool/framework/1.apk`
- Eagle eyed users will notice resources are now decoded before sources now. This is because we need to know the API version via the manifest for decoding the sources

#### Parameter Changes

- Smali/baksmali 2.0 are included. This is a big change from 1.4.2. Please read the smali updates here for more information
- `-o / --output` is now used for the output of apk/directory
- `-t / --tag` is required for tagging framework files
- `-advance / --advanced` will launch advance parameters and information on the usage output
- `-m / --match-original` is a new feature for apk analysis. This retains the apk is nearly original format, but will make rebuild more than likely not work due to ignoring the changes that newer aapt requires
- After `[d]ecode`, there will be new folders (original / unknown) in the decoded apk folder
  - **original** = `META-INF folder` / `AndroidManifest.xml`, which are needed to retain the signature of apks to prevent needing to resign. Used with `-c / --copy-original` on `[b]uild`
  - **unknown** = Files / folders that are not part of the standard AOSP build procedure. These files will be injected back into the rebuilt APK.
- `apktool.yml` collects more information than last version
  - `SdkInfo` - Used to repopulate the sdk information in `AndroidManifest.xml` since newer aapt requires version information to be passed via parameter
  - `packageInfo` - Used to help support Android 4.2 renamed manifest feature. Automatically detects differences between resource and manifest and performs automatic `--rename-manifest-package` on `[b]uild`
  - `versionInfo` - Used to repopulate the version information in `AndroidManifest.xml` since newer aapt requires version information to be passed via parameter
  - `compressionType` - Used to determine the compression that `resources.arsc` had on the original apk in order to replicate during `[b]uild`
  - `unknownFiles` - Used to record name/location of non-standard files in an apk in order to place correctly on rebuilt apk
  - `sharedLibrary` - Used to help support Android 5 shared library feature by automatically detecting shared libraries and using `--shared-lib` on `[b]uild`

#### Examples of new usage in 2.0 vs 1.5.x

| Old (Apktool 1.5.x) | New (Apktool 2.0.x) |
|---|---|
| `apktool if framework-res.apk tag` | `apktool if framework-res.apk -t tag` |
| `apktool d framework-res.apk output` | `apktool d framework.res.apk -o output` |
| `apktool b output new.apk` | `apktool b output -o new.apk` |

### v1.4.x -> v1.5.1

- Update apktool to `v1.5.1`
- Update aapt manually or use package `r05-ibot` via downloading Mac, Windows or Linux
- Remove framework file `$HOME/apktool/framework/1.apk`

## Options

### Utility

Options that can be executed at any time.

`-version, --version`

> *Outputs current version. (Ex: 1.5.2)*

`-v, --verbose`

> *Verbose output. Must be first parameter*

`-q, --quiet`

> *Quiet output. Must be first parameter*

`-advance, --advanced`

> *Advance usage output*

### Decode

These are all the options when decoding an apk.

`--api <API>`

> *The numeric api-level of the smali files to generate (defaults to targetSdkVersion)*

`-b, --no-debug-info`

> *Prevents baksmali from writing out debug info (.local, .param, .line, etc). Preferred to use if you are comparing smali from the same APK of different versions. The line numbers and debug will change among versions, which can make DIFF reports a pain.*

`-f, --force`

> *Force delete destination directory. Use when trying to decode to a folder that already exists*

`--keep-broken-res` - Advanced

> *If there is an error like "Invalid Config Flags Detected. Dropping Resources...". This means that APK has a different structure then Apktool can handle. This might be a newer Android version or a random APK that doesn't match standards. Running this will allow the decode, but then you have to manually fix the folders with -ERR in them.*

`-m, --match-original` - Used for analysis

> *Matches files closest as possible to original, but **prevents** rebuild.*

`-o, --output <DIR>`

> *The name of the folder that apk gets written to*

`-p, --frame-path <DIR>`

> *The folder location where framework files should be stored/read from*

`-r, --no-res`

> *This will prevent the decompile of resources. This keeps the `resources.arsc` intact without any decode. If only editing Java (smali) then this is the recommended action for faster decompile & rebuild*

`-s, --no-src`

> *This will prevent the disassembly of the dex file(s). This keeps the apk `dex` file(s) and simply moves it during build. If you are only editing the resources. This is the recommended action for faster disassemble & assemble*

`-t, --frame-tag <TAG>`

> *Uses framework files tagged via `<TAG>`*

**Rebuild**

---

These are all the options when building an apk.

`-a, --aapt <FILE>`

> *Loads aapt from the specified file location, instead of relying on path. Falls back to `$PATH` loading, if no file found*

`-c, --copy-original` - Will still require signature resign post API18

> *Copies original `AndroidManifest.xml` and `META-INF` folder into built apk*

`-d, --debug`

> *Adds `debuggable="true"` to AndroidManifest file.*

`-f, --force-all`

> *Overwrites existing files during build, reassembling the `resources.arsc` file and `dex` file(s)*

`-o, --output <FILE>`

> *The name and location of the apk that gets written*

`-p, --frame-path <DIR>`

> *The location where framework files are loaded from*