APK逆向专题

2022/9

课程大纲

1.APK基础

2.Dalvik层逆向分析

3.Native层逆向分析

4.修改Apk&调试手段&hook

工具：

1. AndroidStudio(正向分析/开发)
2. APKIDE(拆解APK/逆向分析)
3. GDA(静态逆向分析dalvik层代码)
4. IDA(静态分析native层.so文件)

课程计划

1. 先讲一些理论上的东西
2. 进行操作演示
3. 交给大家自己操作
4. 进行问题分析和总结

说在前面的话

1. 一定要搭好基本的android开发环境！！！
2. 一定要自己动手去操作！！！
3. 有什么问题/感兴趣的点可以及时提出来。

1. APK基础

比赛中的android题目类型，
以及作为安卓逆向人员必须具备的基础知识

APK基础

1.1 android题目类型

CTF 比赛中的Android 题目主要以**APK逆向**为主，一般的出题方式是:**提供一个APK安装程序**，让选手进行逆向和调试分析，从而得出隐藏在其中的flag; 也有可能不直接提供APK安装程序，而是需要通过流量、解密或者拼装等方式获得,但是最终都会获取一个APK文件(或者dex文件)来进行逆向操作。

APK基础

1.1 android题目类型

目前，市面上大部分的Android系统都部署在ARM处理器平台上，CTF 比赛中所出的大部分题目也是基于ARM平台的，因此推荐在做此类题目的时候准备一部Android手机，这样调试起来会比较方便； 当然，模拟器也可以，但是模拟器在性能上会稍微差点且操作烦琐，同时不排除有的APK可能会对模拟器进行验证,徒增烦恼。如果想深人研究Android，推荐使用谷歌的Nexus系列手机,刷机和调试都非常方便。

APK基础

1.1 android题目类型

关于工具：

解答Android题目对于工具的依赖非常强，熟练掌握几款工具能够让你在解题时得心应手。

APK基础

1.2 android基本架构

Android操作系统可以分为4层：

分别是：

Linux内核层

系统运行层

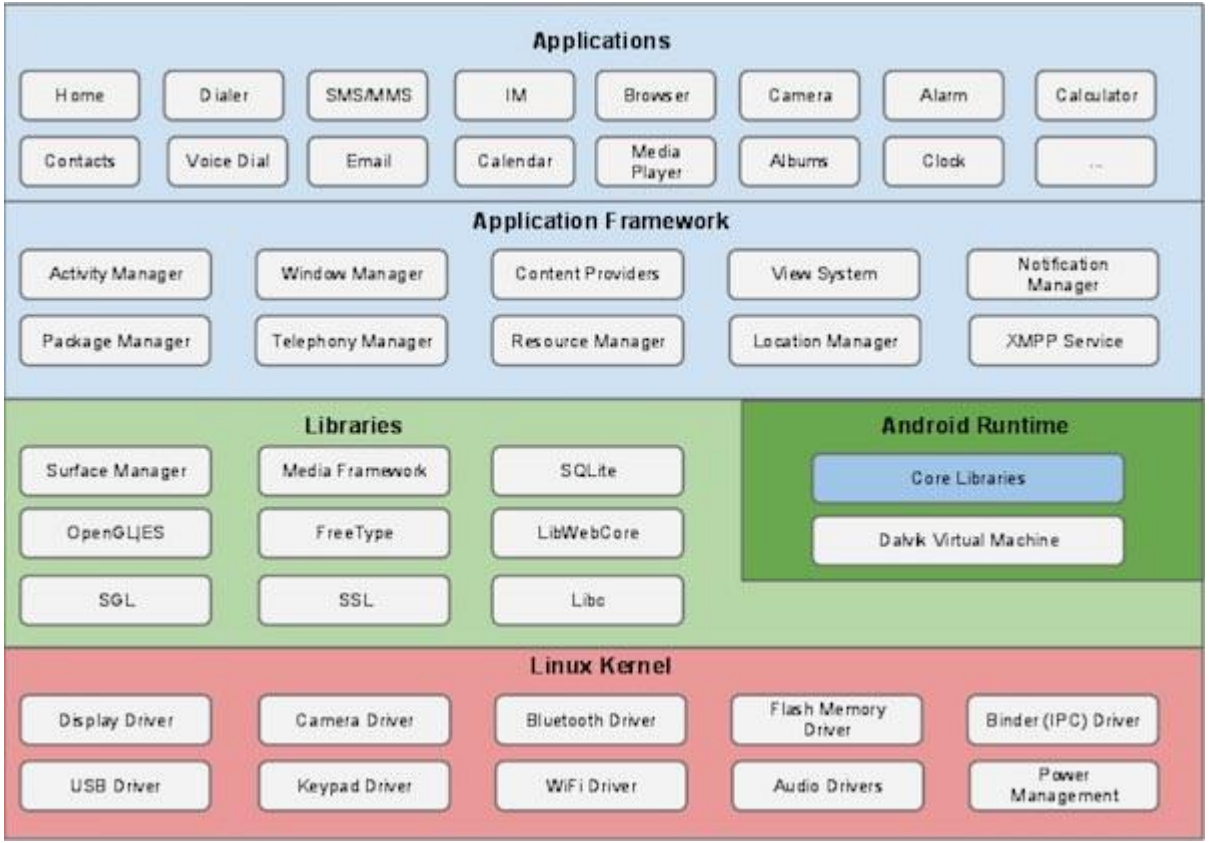
应用框架层

应用层

APK基础

1.2 android基本架构

看图

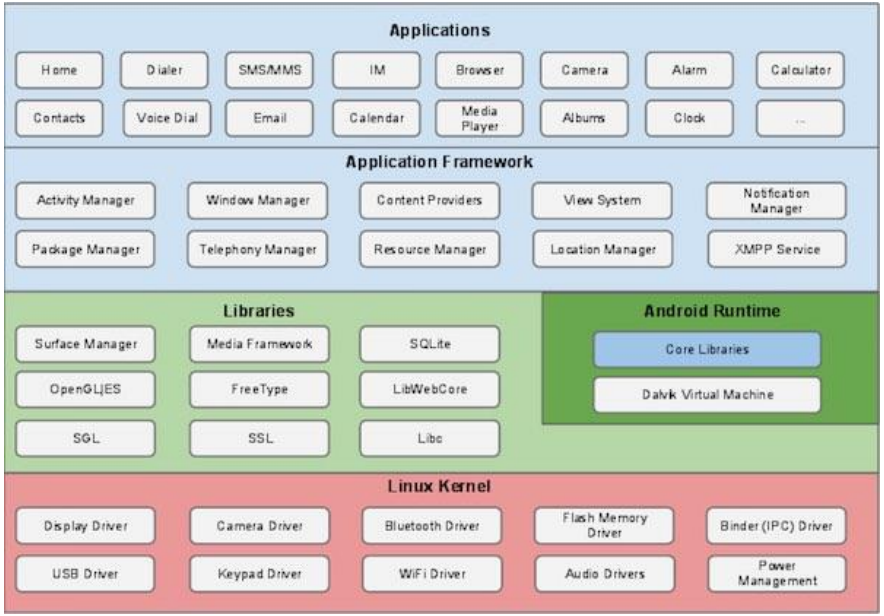


APK基础

1.2 android基本架构

Linux内核

在所有层的最底下是 Linux - 包括大约115个补丁的 Linux 3.6。它提供了基本的系统功能，比如进程管理，内存管理，设备管理（如摄像头，键盘，显示器）。同时，内核处理所有 Linux 所擅长的工作，如网络和大量的设备驱动，从而避免兼容大量外围硬件接口带来的不便。

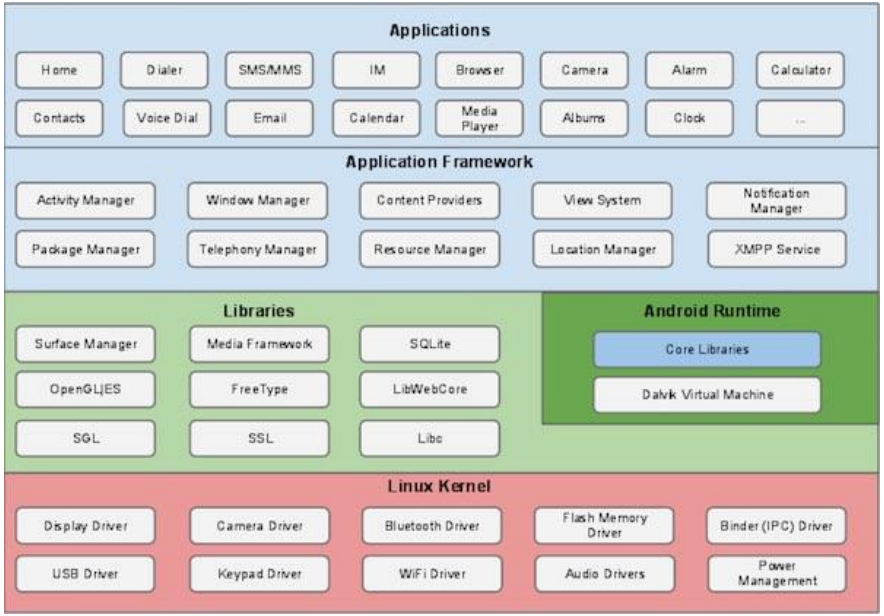


APK基础

1.2 android基本架构

程序库

在 Linux 内核层的上面是一系列程序库的集合，包括开源的 Web 浏览器引擎 Webkit，知名的 libc 库，用于仓库存储和应用数据共享的 SQLite 数据库，用于播放、录制音视频的库，用于网络安全的 SSL 库等。

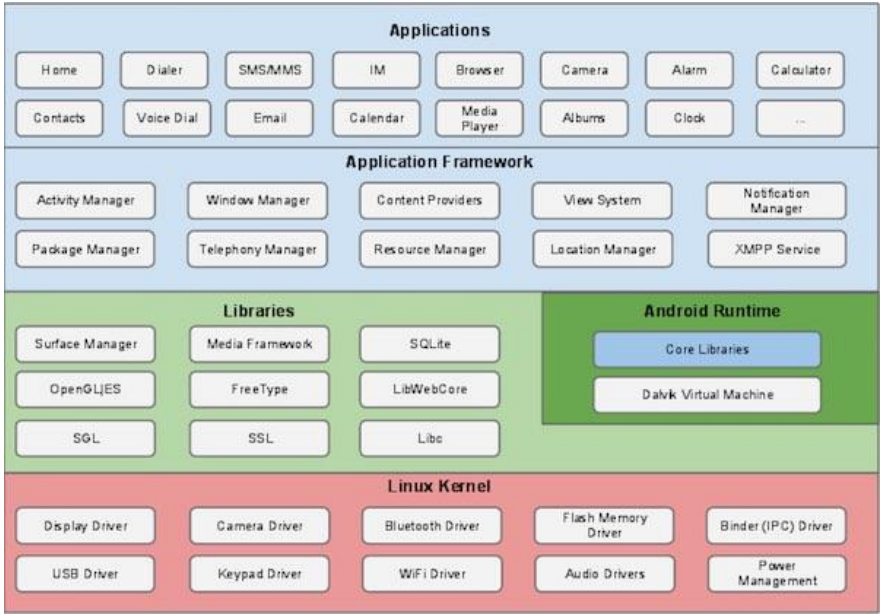


APK基础

1.2 android基本架构

Android程序库

这个类别包括了专门为 Android 开发的基于 Java 的程序库。这个类别程序库的示例包括应用程序框架库，如用户界面构建，图形绘制和数据库访问



APK基础

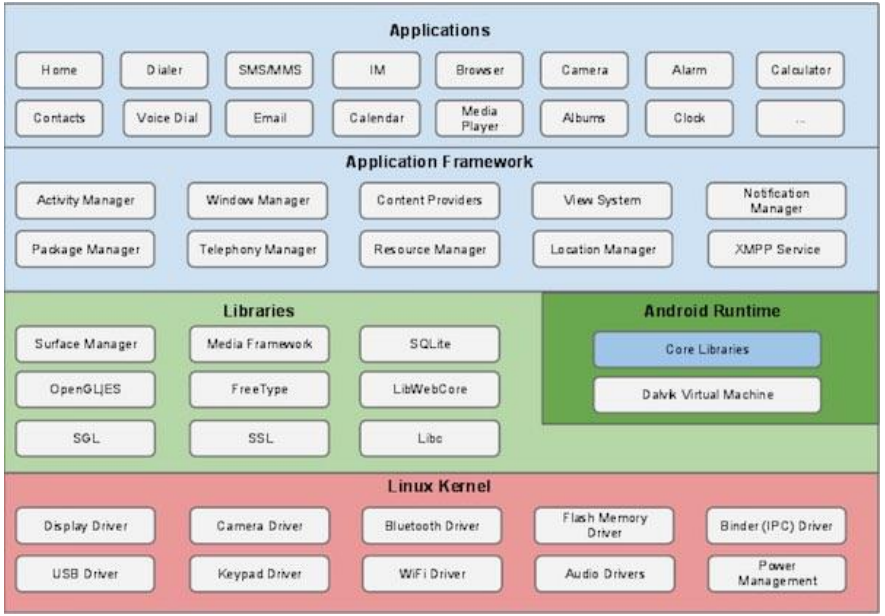
1.2 android基本架构

Android运行时

这是架构中的第三部分，自下而上的第二层。这个部分提供名为 Dalvik 虚拟机的关键组件，类似于 Java 虚拟机，但专门为 Android 设计和优化。

Dalvik 虚拟机使得可以在 Java 中使用 Linux 核心功能，如内存管理和多线程。Dalvik 虚拟机使得每一个 Android 应用程序运行在自己独立的虚拟机进程。

Android 运行时同时提供一系列核心的库来为 Android 应用程序开发者使用标准的 Java 语言来编写 Android 应用程序。

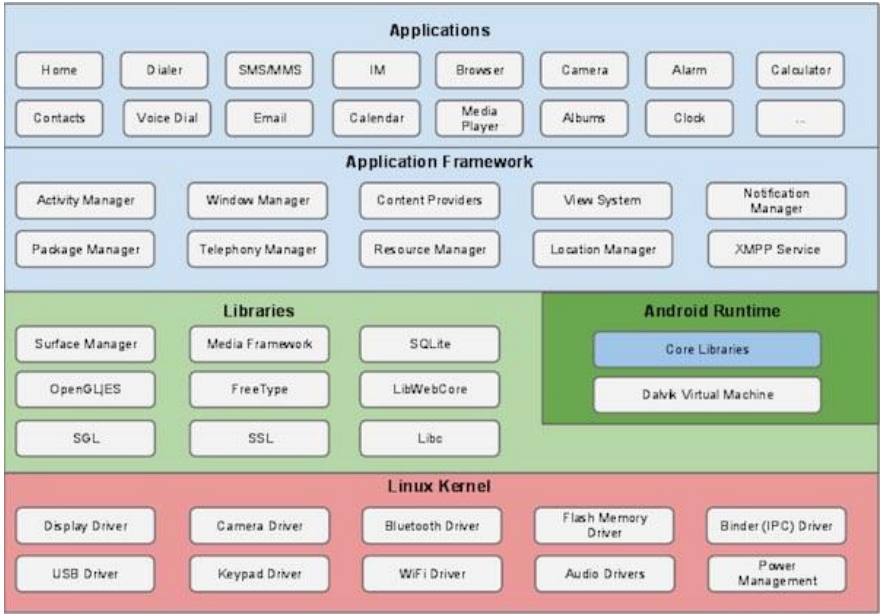


APK基础

1.2 android基本架构

应用框架

应用框架层以 Java 类的形式为应用程序提供许多高级的服务。应用程序开发者被允许在应用中使用这些服务。



APK基础

1.2 android基本架构

从开发人员的角度来看，一个Android应用可以分为两个部分。

一部分使用java实现，也称为Dalvik虚拟机层；

一部分使用c/c++实现，也成Native层；

APK基础

1.2 android基本架构

从出题的角度来看，题目的主要逻辑既可以出在Dalvik层，也可以出在Native层。

对Dalvik层的代码进行逆向操作比较方便，属于比**较简单**的题型；

而对native层的代码进行操作可能会比较复杂，属于**较难**的题型。

APK基础

1.2 android基本架构

关于Dalvik虚拟机

Android应用虽然可以使用Java开发，但是Android应用却不是运行在标准的Java虚拟机上，而是运行在谷歌专门为Android开发的Dalvik虚拟机上。虽然Android从5.0开始默认使用ART虚拟机，抛弃了Dalvik虚拟机，但是Dalvik虚拟机的基础知识仍然是逆向必不可少的，尤其是DEX文件的反编译。

APK基础

1.2 android基本架构

关于Dalvik虚拟机

Dalvik 虚拟机中运行的是Dalvik字节码，并不是Java字节码，所有的Dalvik字节码均由Java字节码转换而来，并打包成一个DEX (DalvikExecutable)可执行文件。**Dalvik虚拟机有一套自己的指令集,以及一套专门的 Dalvik汇编代码。**

APK基础

1.2 android基本架构

关于Native层

Android既可以使用Java开发，也可以与C/C++结合开发，甚至可以使用纯C/C++开发。使用C/C++开发的代码经过编译后会形成一个so文件，会在 Android应用运行时加载到内存中。这与x86平台中Linux加载so库的方式非常相似，唯一不同的是如何将Native层的函数与Dalvik层的函数进行关联，使得Native层的函数在Dalvik层中可以很方便地调用。

APK基础

1.3 ARM架构基础知识

目前绝大多数的Android应用都运行在ARM处理器架构上。

主流的32位ARM处理器架构的版本为ARMv7，64位的ARM处理器的原理与之类似。

目前来说，题目中涉及要了解ARM的都是Native层的逆向，也就是C语言编译的.so，一般来说都会给32位和64位两个版本的.so文件，所以我们选择看的顺眼的去分析就可以了。

APK基础

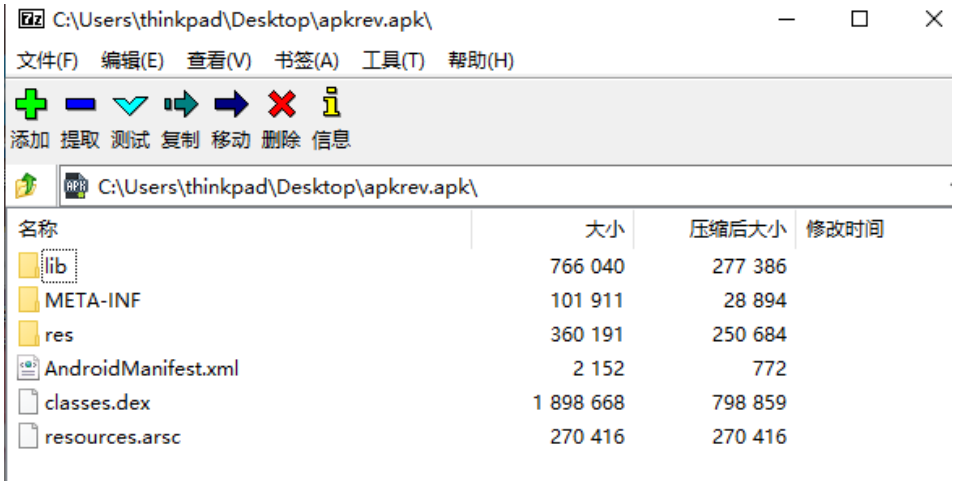
1.3 ARM架构基础知识

所以这部分内容我们讲到Native层逆向的时候再去详细的说~

APK基础

1.4 APK文件格式

APK文件其实是一个ZIP压缩文件，使用任何压缩软件基本上都可以直接解压。



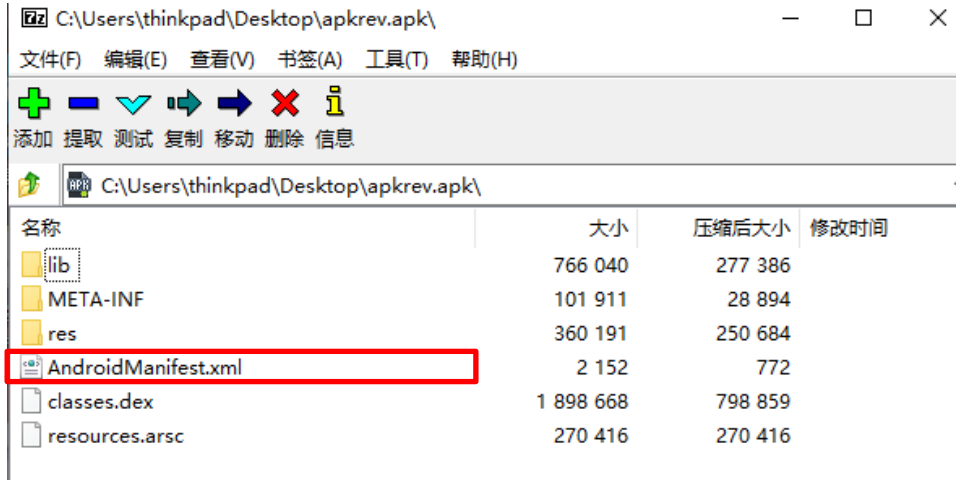
APK基础

1.4 APK文件格式

1. AndroidManifest.xml

是这个APK的属性文件，所有的APK都需要包含这个文件，这个文件写明了这个APK所具有的Activity，所需要的函数，启动类是那一个，需要什么权限等信息。

当然直接打开解压后这个文件会是乱码，需要用工具去解析



APK基础

1.4 APK文件格式

1. AndroidManifest.xml

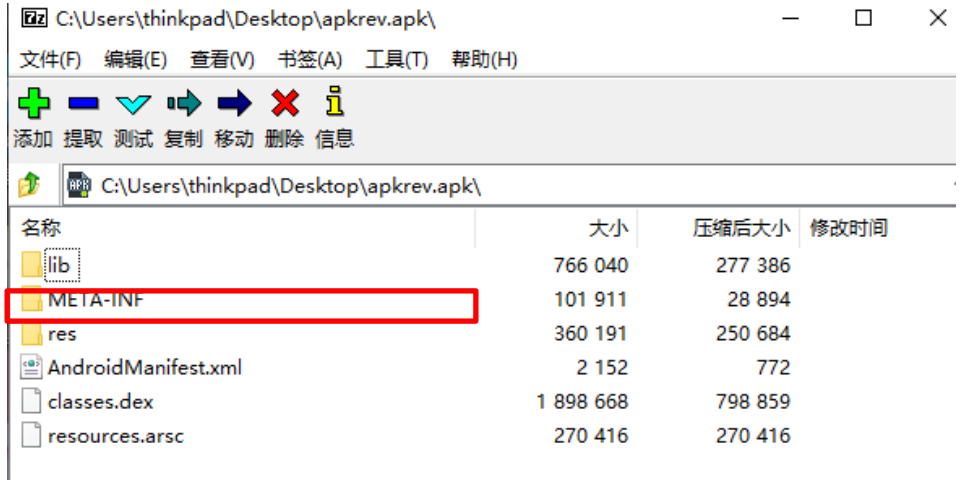
```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.test4">

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/icon"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/icon"
        android:supportsRtl="true"
        android:theme="@style/AppTheme"
        android:usesCleartextTraffic="true">

        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>


</manifest>
```

APK基础

1.4 APK文件格式

2. META-INF文件夹

这个文件夹是编译过程中自动生成的文件夹尽量不要手动修改。

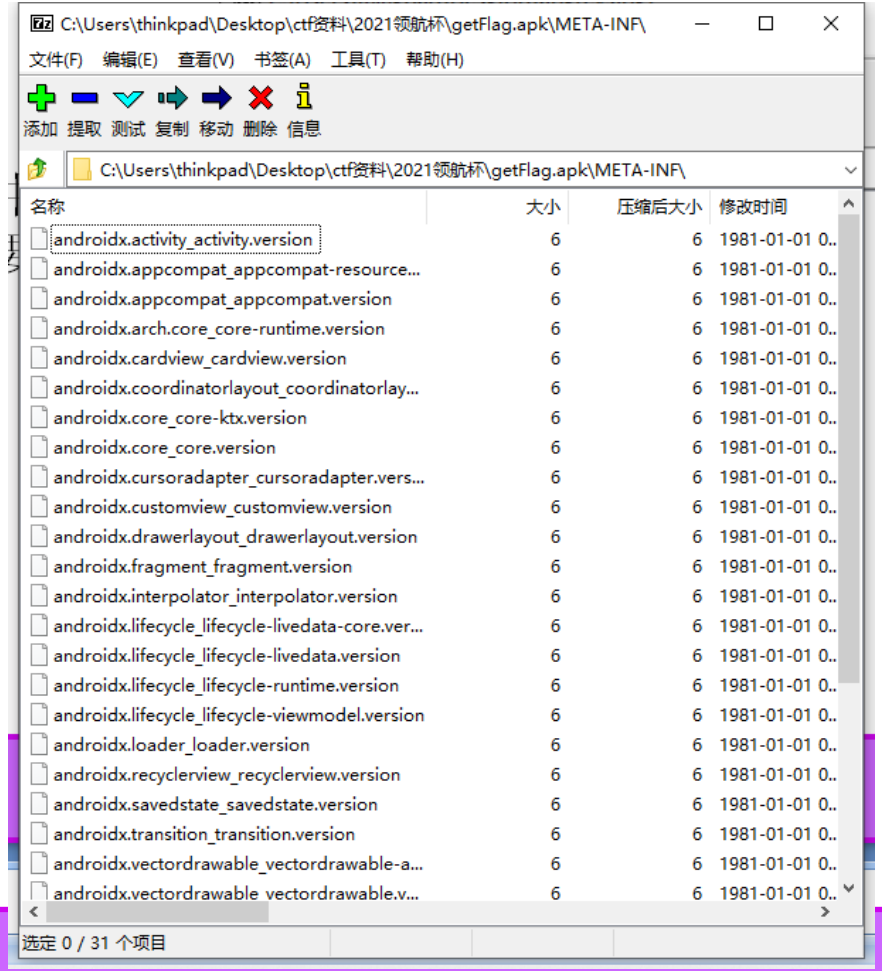


APK基础

1.4 APK文件格式

2. META-INF文件夹

这个文件夹是编译过程中自动生成的文件夹尽量不要手动修改。

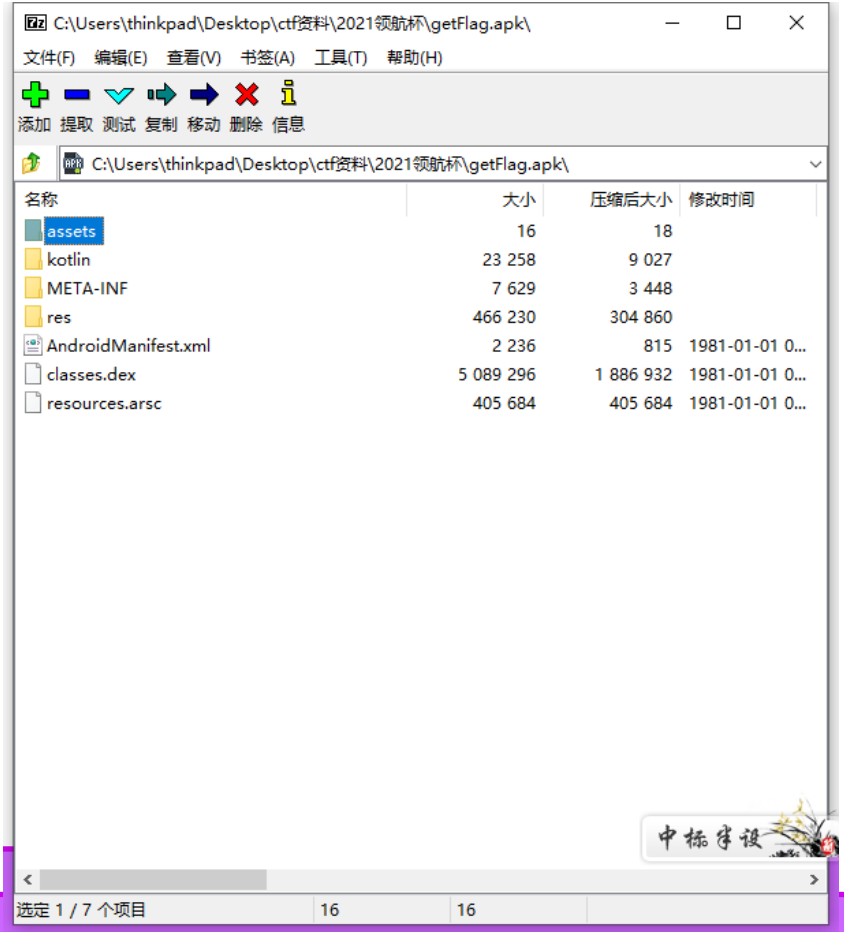


APK基础

1.4 APK文件格式

3. Assets文件夹

存放在这个文件夹里的文件或原封不动的打包到APK里面，因此这个文件夹里经常会存放一些程序中会使用的文件，比如说程序的解密秘钥或者加密后的密文等等。

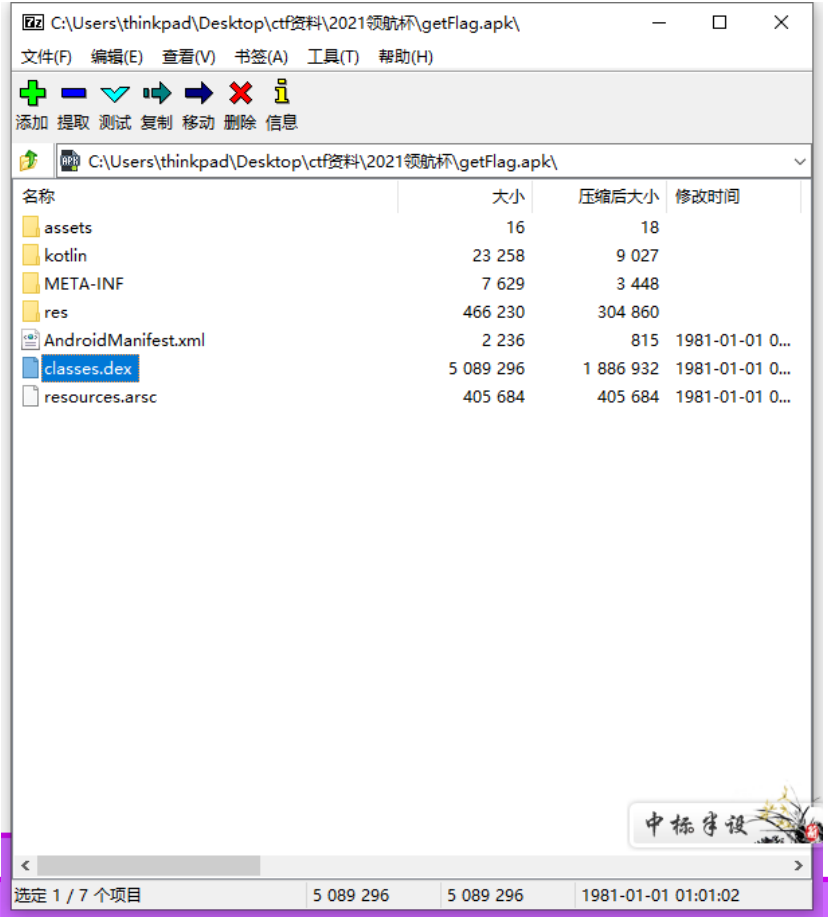


APK基础

1.4 APK文件格式

4. classes.dex

这个就是存放Dalvik字节码的DEX文件。

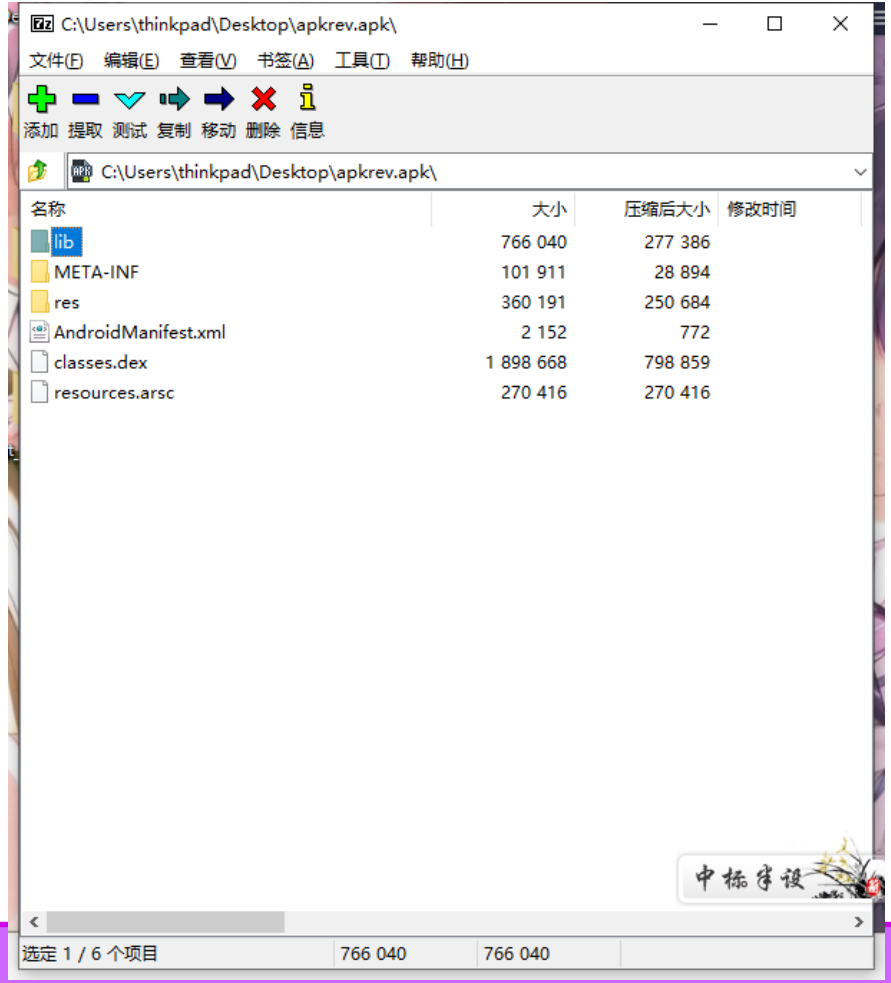


APK基础

1.4 APK文件格式

5. Libs文件夹

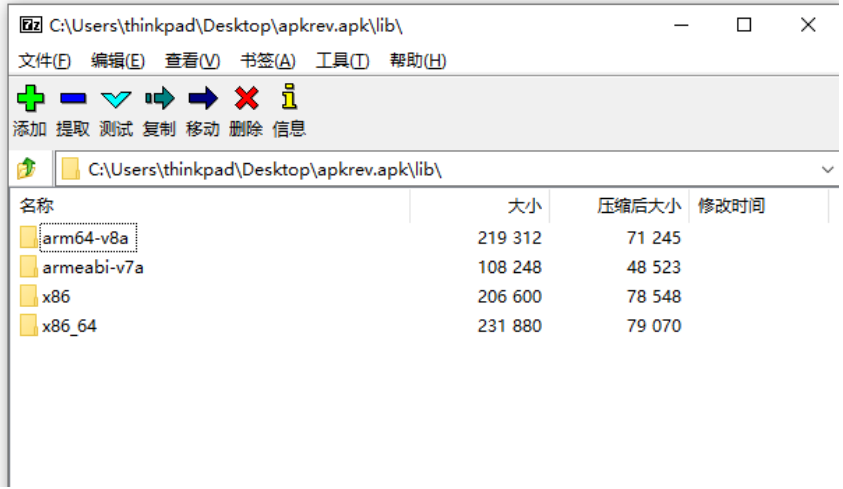
这个文件夹包含Native层所需要的lib库，一般为libxxx.so格式，lib文件夹中可以包含多个lib文件。

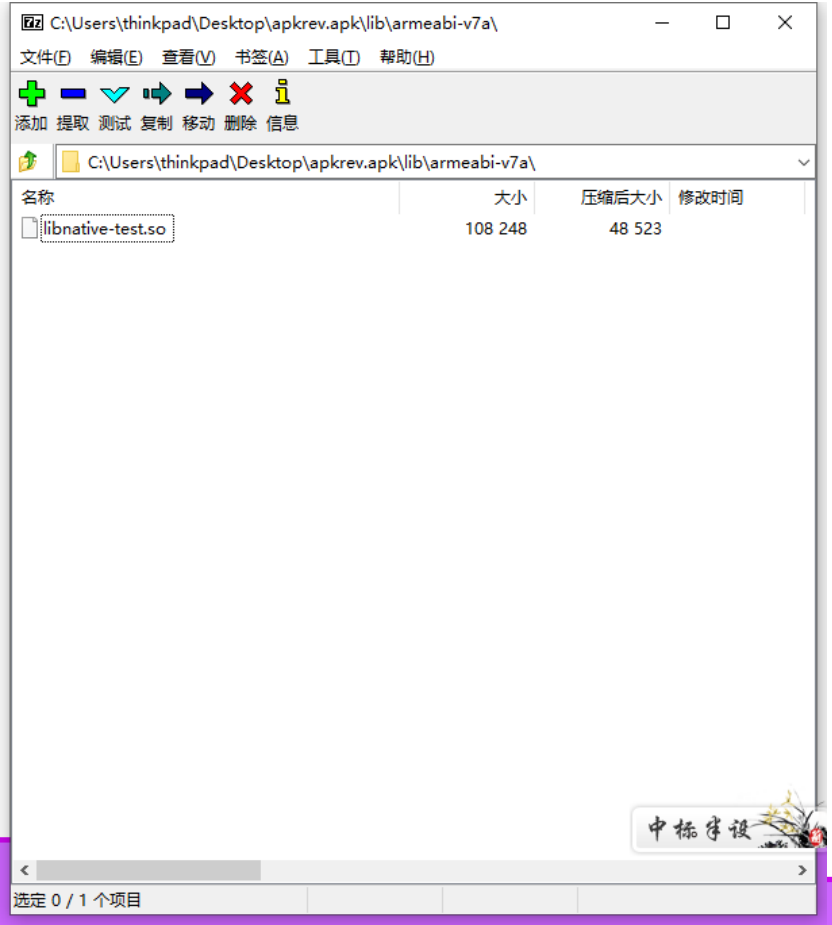


APK基础

1.4 APK文件格式

5. Libs文件夹





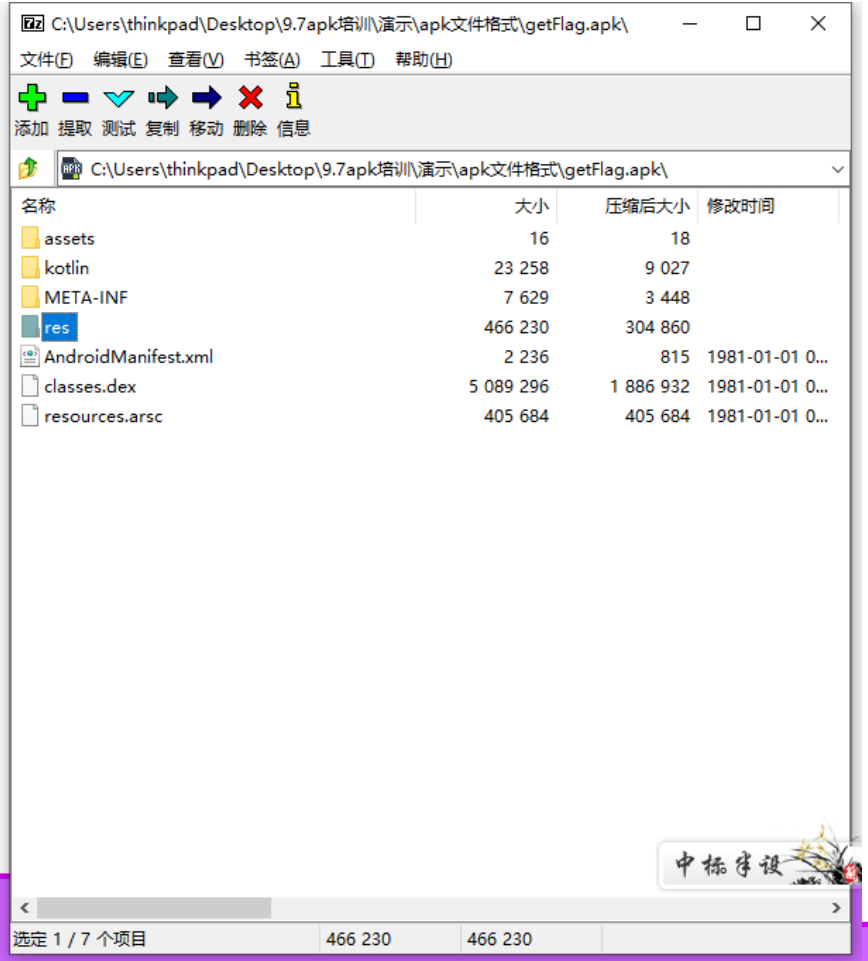
APK基础

1.4 APK文件格式

6. Res文件夹

存放了资源、字体的相关文件，比如说位图。

【演示】

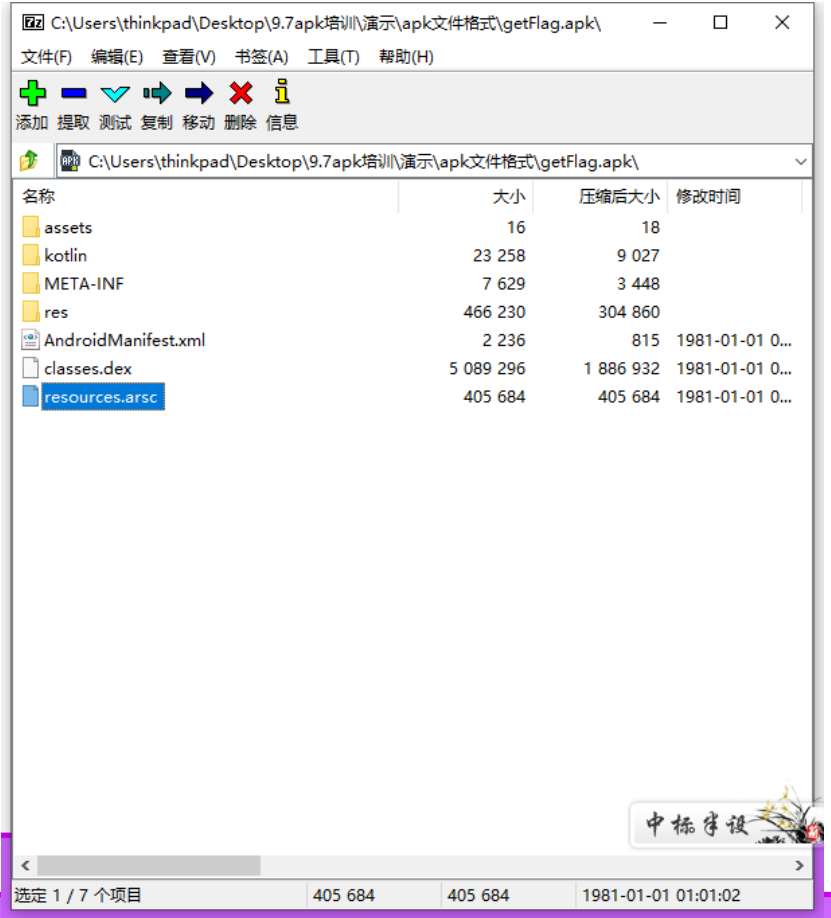


APK基础

1.4 APK文件格式

7. Resources.arsc

文件里存放了apk中所使用的资源的名字、ID、类型等信息，若直接使用编辑器打开看到的会是乱码。



APK基础

1.5 ADB

ADB（android debug bridge）是谷歌官方提供的命令行工具，用来连接真机或者模拟器。

只要在相应的android系统中打开usb调试，就可以使用ADB连接手机，ADB最主要的功能是查看连接的手机、打开一个shell，查看日志、上传和下载文件。

APK基础

1.5 ADB

Androidstudio中SDK中的位置：

SDK\sdk\platform-tools\adb.exe

APK基础

1.5 ADB

ADB的使用【简单演示】

1. 查看连接的手机或者模拟器：adb devices
2. 安装apk： adb install <apk路径> / adb install -t *.apk
3. 卸载app：adb uninstall <package>
4. 打开shell：adb shell
5. 查看日志：adb logcat 、adb logcat –s [tag]
6. 上传文件：adb push xxx /data/local/tmp
7. 下载文件：adb pull /data/local/tmp/some_file　some_location
8. 将本地端口转发到远程设备的端口：adb forward LOCAL REMOTE
9. 列出所有的转发端口：adb forward –list
10. 将远程设备的端口转发到本地： adb reverse REMOTE LOCAL
11. 列出所有反向端口转发：adb reverse --list

2. Dalvik层逆向

Java这一层代码的相关~

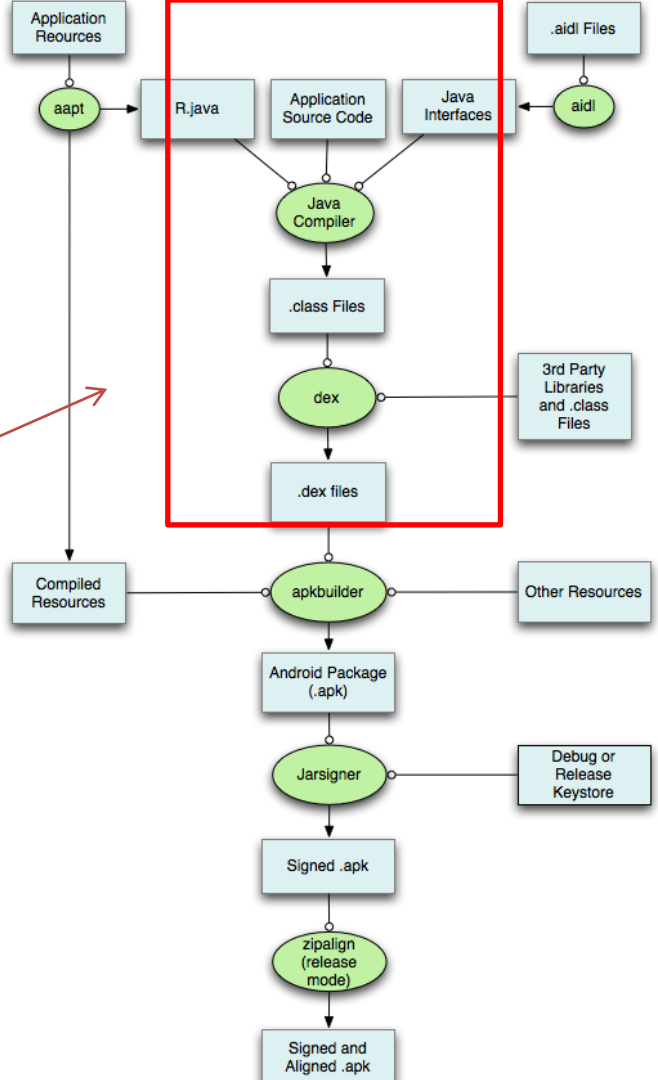
2.1 还是一些基础知识

DALVIK层逆向

2.1 还是一些基础知识

先补充一个的APK打包流程

DALVIK逆向中主要关注的部分



DALVIK层逆向

2.1 还是一些基础知识

在执行 Android Java 层的代码时，其实就是 Dalvik(ART) 虚拟机（使用 C 或 C++ 代码实现）在解析 Dalvik 字节码**【在classes.dex文件中】**，从而模拟程序的执行过程。

自然，Dalvik 字节码晦涩难懂，研究人员们给出了 Dalvik 字节码的一种助记方式：**smali 语法**。**【相当于汇编和机器码的关系】**
通过一些工具（如 apktool），我们可以把已有的 dex 文件转化为若干个 smali 文件（**一般而言，一个 smali 文件对应着一个类**），然后进行阅读。对于不同的工具来说，其转换后的 smali 代码一般都不一样，毕竟这个语法不是官方的标准。这里我们介绍比较通用的语法。值得注意的是，在 smali 语法中，使用的都是寄存器，但是其在解释执行的时候，很多都会映射到栈中。

这里我们使用的是APKIDE来进行分析。

DALVIK层逆向

2.1 还是一些基础知识

*下面讲的这部分在初学阶段不需要深入理解，简单了解就可以了

一个 Smali 文件的基本信息如下

基本类信息

前三行描述转换为该 Smali 文件的类的信息

如果类实现了接口，对应的接口信息

如果类使用了注解，对应的注解信息

字段描述

方法描述

【演示】

DALVIK层逆向

2.1 还是一些基础知识

比较有意思的是，Smali 代码基本上还原了 java 代码中含义。它主要有以下两种类型的语句：
声明语句用来声明 java 中自顶向下的类，方法，变量类型，以及每个方法中所要使用的寄存器的个数等信息。
执行语句来执行 java 中的每一行代码，包含方法的调用，字段的读写，异常的捕捉等操作。

整体来说，Smali 代码的可读性还是比较强的。

DALVIK层逆向

2.1 还是一些基础知识

声明语句

在 smali 代码中，声明语句一般都是以 . 开始。

DALVIK层逆向

2.1 还是一些基础知识

寄存器 ¶

目前，Dalvik 使用的寄存器都是 32 位，对于 64 位类型的变量，如 double 类型，它会使用两个相邻的 32 位寄存器来表示。

此外，我们知道 Dalvik 最多支持 65536 个寄存器 (编号从 0~65535)，但是 ARM 架构的 cpu 中只有 37 个寄存器。那 Dalvik 是怎么做的呢？其实，每个 Dalvik 虚拟机维护了一个调用栈，该调用栈用来支持虚拟寄存器和真实寄存器相互映射的。

DALVIK层逆向

2.1 还是一些基础知识

寄存器声明 ¶

在执行具体方法时，Dalvik 会根据 .registers 指令来确定该函数要用到的寄存器数目，虚拟机会根据申请的寄存器的数目来为该方法分配相应大小的栈空间，dalvik 在对这些寄存器操作时，其实都是在操作栈空间。

DALVIK层逆向

2.1 还是一些基础知识

寄存器命名规则 ¶

一个方法所申请的寄存器会分配给函数方法的参数 (parameter) 以及局部变量 (local variable) 。在 smali 中，一般有两种命名规则

v 命名法

p 命名法

【演示】

DALVIK层逆向

2.1 还是一些基础知识

假设方法申请了 m+n 个寄存器，其中局部变量占 m 个寄存器，参数占 n 个寄存器，对于不同的命名规则，其相应的命名如下

属性	v 命名法	p 命名法
局部变量	$v_0, v_1, \ldots, v_{m-1}$	$v_0, v_1, \ldots, v_{m-1}$
函数参数	$v_m, v_{m+1}, \ldots, v_{m+n}$	$p_0, p_1, \ldots, p_{n-1}$

一般来说我们更倾向于 p 命名法，因为其具有较好的可读性，可以方便地让我们知道寄存器属于哪一种类型。

而这其实也就是 smali 语法中常见的寄存器命名规则，p 开头的寄存器都是参数寄存器，v 开头的寄存器都是局部变量寄存器，两者的数量之和为方法申请的寄存器数量。

DALVIK层逆向

2.1 还是一些基础知识

变量类型

在 Dalvik 字节码中，变量主要分为两种类型

类型	成员
基本类型	boolean，byte，short，char，int，long，float，double，void（只用于返回值类型）
引用类型	对象，数组

DALVIK层逆向

2.1 还是一些基础知识

【演示】

但是，我们在 smali 中其实并不需要把一个变量的类型的描述的全称全部放进去，我们只需要可以识别它即可，那我们可以怎么做呢？可以对它进行简写啊。dalvik 中简写方式如下

java 类型	类型描述符
boolean	Z
byte	B
short	S
char	C
int	I
long	J
float	F
double	D
void	V
对象类型	L
数组类型	[

DALVIK层逆向

2.1 还是一些基础知识

其中对象类型可以表示 Java 代码中的所有类。比如说如果一个类在 java 代码中的以 package.name.ObjectName（全名）的方式被引用，那么在 Davilk 中，其描述则是 `Lpackage/name/ObjectName;` ，其中

- L 即上面所说的对象类型。
- 全名中的 `.` 被替换为 `/` 。
- 后面跟了一个 `;` 。

比如说在 `java.lang.String` ，其相应的形式为 `Ljava/lang/String;`

> 注：所谓全名就是它的全程不仅仅是简写，比如 String 其实是 java.lang.String。

DALVIK层逆向

2.1 还是一些基础知识

数组类型可以表示 java 中的所有数组。其一般的构成形式由前向后依次分为两个部分

- **数组维数**个 [，但数组的维数最多为 255。
- 数据元素类型，这里的类型自然就不能是 [了。

比如说 int 数组 `int []` 在 smali 中的表示形式为 `[I` 。

比如说数组类型 `String[][]` 在 smali 中的表示形式为 `[[Ljava/lang/String;` 。

DALVIK层逆向

2.1 还是一些基础知识

字段

在 java 的类中，一般都会有成员变量，也称为其属性或者字段。java 中的字段分为

- 普通字段，实例属性
- 静态字段，类属性，所有的类实例共享。

DALVIK层逆向

2.1 还是一些基础知识

```
# instance fields
.field public en:Lcom/nepnep/app/Encrypt;
```

普通字段

声明如下

```
#instance fields
.field <访问权限修饰符> [非权限修饰符] <字段名>:<字段类型>
```

其中访问权限修饰符可以为

- public
- private
- protected

非权限修饰符可以为 (**查明其用法!!!**)

- final
- volidate
- transient

DALVIK层逆向

2.1 还是一些基础知识

举个例子，如下

```
# instance fields
.field private str1:Ljava/lang/String;
```

这里声明其实如下

```
private java.lang.String str1;
```

DALVIK层逆向

2.1 还是一些基础知识

静态字段

一般表示如下

```
#static fields
.field <访问权限> static [修饰词] <字段名>:<字段类型>
```

这里我们就不介绍相应内容了，直接给出一个例子

```
# static fields
.field public static str2:Ljava/lang/String;
```

其实声明如下

```
public static java.lang.String str2;
```

DALVIK层逆向

2.1 还是一些基础知识

```
uctor <clinit>()V

"native-lib"

}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V
```

方法

在 smali 代码中，方法一般的展现形式如下

```
# 描述方法类型
.method <访问权限修饰符> [修饰符] <方法原型>
      <.locals>
      [.parameter]
      [.prologue]
      [.line]
      <代码逻辑>
      [.line]
      <代码逻辑>
.end
```

其中第一行以注释形式描述方法的类型，一般是反编译工具添加上去的，分为两种类型

- 直接方法，direct method
- 虚方法，virtual method

访问权限可能有有以下形式，与 java 中的一一对应

- public
- private
- protected

DALVIK层逆向

2.1 还是一些基础知识

关于smali以及smali的混淆还有很多很多的内容，更多详细的内容可以去ctf-wiki上面更加深入的了解~

https://ctf-wiki.org/android/basic_operating_mechanism/java_layer/smali/smali/

2.2 关于工具

DALVIK层逆向

2.2 关于工具

当然，我们之前也说过APK的打包流程，从java代码到.class文件到.dex的流程。

那么其实工具就可以完成从.dex到.jar(.class文件的集合)的过程，我们只要直接分析.class文件就可以了。

经典的.class反编译工具：jd-gui，在APKIDE中已经自带了。
【演示+自己操作】

DALVIK层逆向

2.2 关于工具

题目名称：Crackme1

描述：使用apkide对该程序进行逆向，得到正确输入的flag
Flag格式：flag{xxxx}

DALVIK层逆向

2.2 关于工具

GDA的使用：

GDA是现在最方便的Dalvik层静态反编译工具之一
非常快捷！直接反编译出java代码。

【用Crackme1演示】

2.3 正向分析

DALVIK层逆向

2.3 正向分析

一切又要从HelloWorld说起……【用Crackme1演示】

重点：

1. 理解apk运行的基本流程(类似于MVC)
2. 理解AndroidManifest.xml的一些相关属性
3. 理解R资源的相关内容(R.id等等)
4. 理解相应的Button监听器回调(Listener)

DALVIK层逆向

2.3 正向分析

Android开发环境的搭建

可以参考： https://blog.csdn.net/u014720022/article/details/93320488

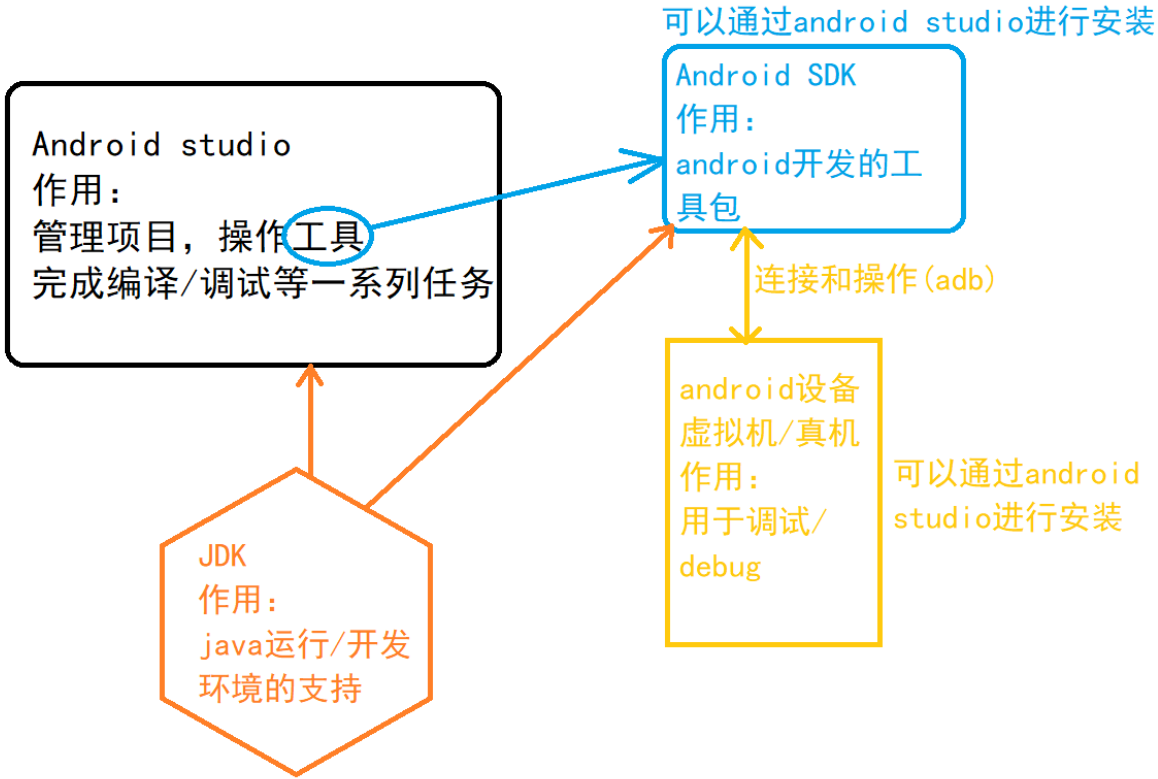
步骤：

1. 配置jdk(java的环境)
2. 配置android studio的环境(在android studio的环境中包含了android的sdk)
3. 配置调试用的android设备(虚拟机/真机)

DALVIK层逆向

2.3 正向分析

Android开发环境的搭建



DALVIK层逆向

2.3 正向分析

ProGuard混淆

ProGuard混淆是Android SDK默认自带的混淆器，其主要功能是对类名、方法名、变量名等标识符进行混淆，将他们修改为无意义的字母组合。

能够加大一点点(真的只有一点点)分析难度。
【演示(Crackme2)】

DALVIK层逆向

2.3 正向分析

题目名称：Crackme2

描述：使用apkide或者GDA对该程序进行逆向，得到正确输入的flag
Flag格式：flag{xxxx}

DALVIK层逆向

2.3 正向分析

题目名称：Crackme2

涉及到的知识点：

1. Proguard混淆
2. Xor运算

DALVIK层逆向

2.3 正向分析

关于Dalvik层面的混淆还有很多很多种方式，如DEX破坏、APK伪加密、APK增加数据、DEX隐藏等等，在搞清楚原理以后都能比较容易的解决。

【这个在之后会有更深入的讲解】

2.4 真题练习

DALVIK层逆向

2.4 真题练习

题目名称：Crackme3

描述：对题目进行逆向分析，得到正确输入的flag

DALVIK层逆向

2.4 真题练习

知识点补充：

Zip伪加密
这个考点常出现在misc类型的题目中，而apk文件刚好是zip文件格式，所以也存在加密位，从而出现伪加密的防破解方式。

【几乎没有遇到过这种情况】

DALVIK层逆向

2.4 真题练习

题目名称： easystr.apk

描述：
对题目进行逆向分析，得到正确输入的flag

DALVIK层逆向

2.4 真题练习

题目名称：Crackme4.apk

描述：对题目进行逆向分析，得到正确输入的flag

Hint：可能需要使用爆破的方式哦。

DALVIK层逆向

2.4 真题练习

题目名称：getFlag.apk

描述：
【题目选自2021年江苏省领航杯】
对题目进行逆向分析，得到正确输入的flag

DALVIK层逆向

2.4 真题练习

知识点补充：

加壳与加载DEX
本质上就是有一个以上的dex文件，将主要的代码逻辑隐藏在了另外一个dex文件之中，通过DexClassLoader的方式进行加载。

- 用于加载包含*.dex文件的jar包或apk文件
- 要求一个应用私有可写的目录去缓存编译的class文件
- 不允许加载外部存储空间的文件，以防注入攻击

DALVIK层逆向

2.4 真题练习

题目名称：cm1.apk

描述：
题目选自VNCTF2022 cm1
对题目进行逆向分析，得到正确输入的flag

Hint：真正的dex在哪里呢？
此题稍微有一点点复杂，如果有疑问或者卡住了可以提出来~

3. Native层逆向

c/c++这一层代码的相关~

3.1 正向分析

NATIVE层逆向

3.1 正向分析

Native层的破解是比赛中的重点，也是比赛中中等题和难题必定要涉及的知识点。

NATIVE层逆向

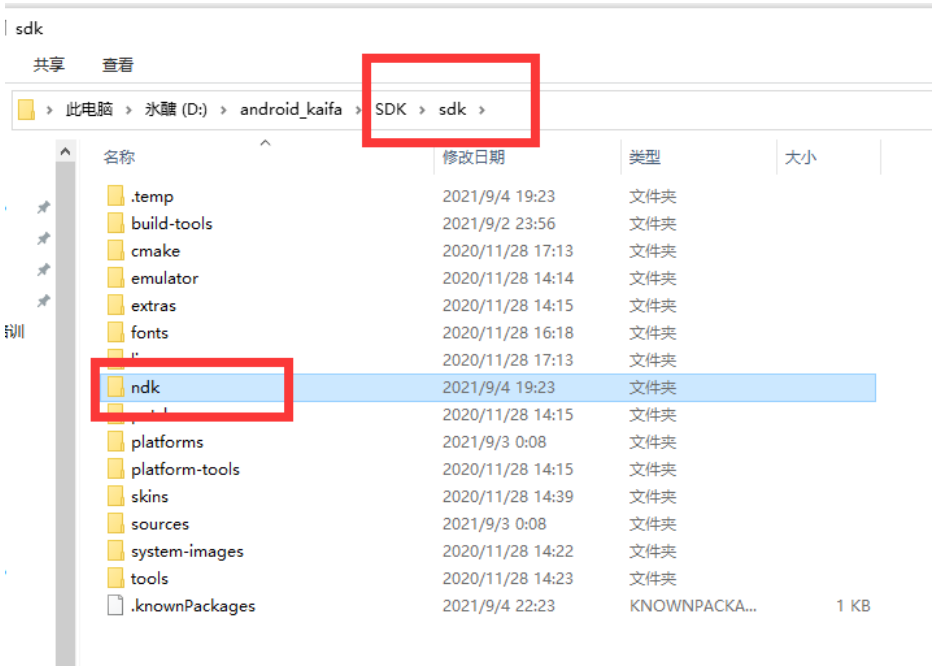
3.1 正向分析

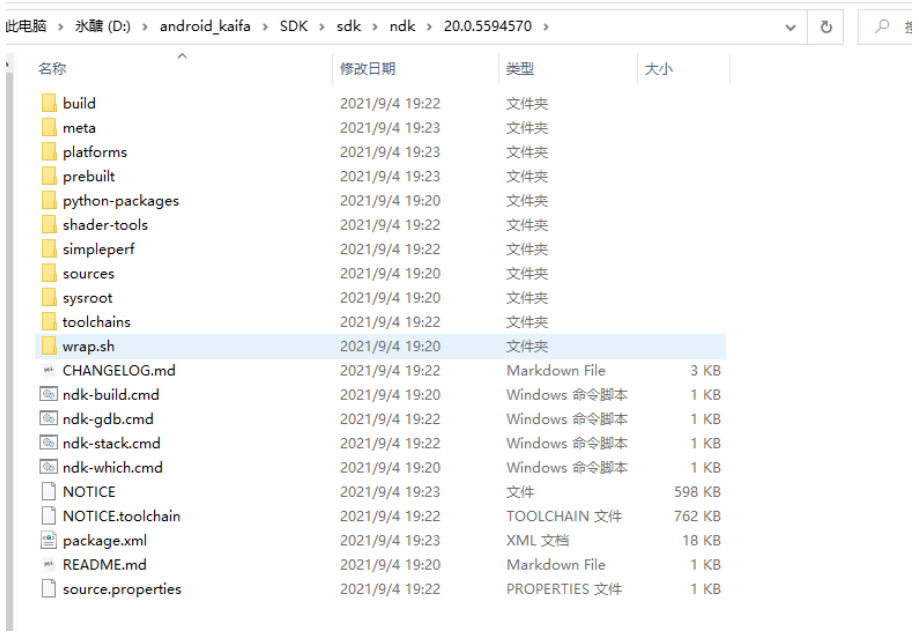
NDK(Native Development Kit)是谷歌提供的一个开发工具包，使用这个工具能够让Java层的代码调用到Native层，也就是c/c++代码，使apk能够实现一些更底层的功能。目前NDK支持x86,x86-64,ARM,mip等架构。

NDK可以使用AndroidStudio直接安装。

NATIVE层逆向

3.1 正向分析





NATIVE层逆向

3.1 正向分析

NDK开发的目的是为了使Java层能够调用c/c++层中的某个函数，而不是使Native层独立于Java层运行，而NDK正式提供了这样一种方法，使得Java层的方法能够与c/c++层的函数能够耦合起来，让调用c/c++层的函数与调用普通java方法一样简单。

【在java中调用C函数】

NATIVE层逆向

3.1 正向分析

在Native层中注册函数有两种基本方法：

1. 静态注册
2. 动态注册

NATIVE层逆向

3.1 正向分析

1. 静态注册

1、首先我们需要在Java代码中声明Natvie方法原型

```
public native void helloJNI(String msg);
```

2、其次我们需要在C/C++代码里声明JNI方法的原型

如：

```
extern "C"
JNIEXPORT void JNICALL
Java_com_kgdwbb_jnistudy_MainActivity_helloJNI(JNIEnv* env, jobject thiz,jstring msg) {
    //do something
}
```

NATIVE层逆向

3.1 正向分析

1. 静态注册

2、其次我们需要在C/C++代码里声明JNI方法的原型

如：

```
extern "C"
JNIEXPORT void JNICALL
Java_com_kgdwbb_jnistudy_MainActivity_helloJNI(JNIEnv* env, jobject thiz,jstring msg) {
    //do something
}
```

1. extern “C”。JNI函数声明声明代码是用C++语言写的，所以需要添加extern “C”声明；如果源代码是C语言声明，则不需要添加这个声明。
2. JNIEXPORT。**这个关键字表明这个函数是一个可导出函数**。每一个C/C++库都有一个导出函数列表，只有在这个列表里面的函数才可以被外部直接调用，类似Java的public函数和private函数的区别。**【相当于添加到export table导出表】**
3. JNICALL。说明这个函数是一个JNI函数，用来和普通的C/C++函数进行区别。

NATIVE层逆向

3.1 正向分析

1. 静态注册

2、其次我们需要在C/C++代码里声明JNI方法的原型

如：

```
extern "C"
JNIEXPORT void JNICALL
Java_com_kgdwbb_jnistudy_MainActivity_helloJNI(JNIEnv* env, jobject thiz,jstring msg) {
    //do something
}
```

4. Void 返回值类型

5. JNI函数名原型：Java_ + JNI方法所在的完整的类名，把类名里面的”.”替换成”_” + 真实的JNI方法名，这个方法名要和Java代码里面声明的JNI方法名一样。

6. env 参数 是一个执行JNIENV函数表的指针。

7. thiz 参数 代表的是声明这个JNI方法的Java类的引用。

8. msg 参数就是和Java声明的JNI函数的msg参数对于的JNI函数参数

NATIVE层逆向

3.1 正向分析

1. 静态注册

Java和JNI基本类型对照表

Java类型	JNI类型	C/C++类型	大小
Boolean	jboolean	unsigned char	无符号8位
Byte	jbyte	char	有符号8位
Char	jchar	unsigned short	无符号16位
Short	jshort	short	有符号16位
Integer	jint	int	有符号32位
Long	jlong	long long	有符号64位
Float	jfloat	float	32位浮点值
Double	jdouble	double	64位双精度浮点值

NATIVE层逆向

3.1 正向分析

1. 静态注册

Java和JNI引用类型对照表

与Java基本类型不同，引用类型对开发人员是不透明的。Java类的内部数据结构并不直接向原生代码公开。也就是说原生C/C++代码并不能直接访问Java代码的字段和方法。

Java类型	C/C++类型
java.lang.Class	jclass
java.lang.Throwable	jthrowable
java.lang.String	jstring
java.lang.Object	jobject
java.util.Objects	jobjects
java.lang.Object[]	jobjectArray
Boolean[]	jbooleanArray
Byte[]	jbyteArray
Char[]	jcharArray
Short[]	jshortArray
int[]	jintArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
通用数组	jarray

NATIVE层逆向

3.1 正向分析

1. 静态注册

【演示 CrackmeN1】

NATIVE层逆向

3.1 正向分析

2. 动态注册

```
/**
 * 默认会调用的方法，这里把方法注册好
 **/
JNIEXPORT jint JNI_OnLoad(JavaVM *vm, void *reserved) {
    //OnLoad方法是没有JNIEnv参数的，需要通过vm获取。
    JNIEnv *env = NULL;
    if (vm->AttachCurrentThread(&env, NULL) == JNI_OK) {
        //获取对应声明native方法的Java类
        jclass  clazz = env->FindClass(JNI_CLASS);
        if (clazz == NULL) {
            return JNI_FALSE;
        }
        //注册方法，成功返回正确的JNIVERSION。
        if (env->RegisterNatives(clazz, method_table,
sizeof(method_table)/ sizeof(method_table[0]))==JNI_OK) {
            return JNI_VERSION_1_4;
        }
    }
    return JNI_FALSE;
}
```

```
#include <stddef.h>
#include "jni.h"

#define JNI_CLASS "com/simple/dynamicdemo/NdkTest" //定义native方法的java文件

/**
 * 返回一个字符串
 * **/
JNIEXPORT jstring JNICALL native_hello(JNIEnv *env, jclass clazz) {
    return env->NewStringUTF("Hello from C++");
}

/**
 * 求两个int的值
 * **/
JNIEXPORT jint JNICALL native_add(JNIEnv *env, jobject object, jint a, jint b) {
    return a + b;
}

/**
 * 方法数组，JNINativeMethod的第一个参数是Java中的方法名，第二个参数是函数签名，第三个参数是对应的方法指针，
 * java方法的签名一定要与对应的C++方法参数类型一致，否则注册方法可能失败
 * **/
static JNINativeMethod method_table[] = {
        {"native_hello", "()Ljava/lang/String;", (void *) native_hello},
        {"native_add",   "(II)I",            (void *) native_add}
};
```

NATIVE层逆向

3.1 正向分析

2. 动态注册

【演示 CrackmeN3】

3.2 逆向分析

NATIVE层逆向

3.2 逆向分析

已经有了正向编程的基础，更加有利于我们进行Native层的逆向分析。

主要按两个方面去关注：

1. Java类中调用了什么Native层的方法？
2. Native层中是通过**静态注册**还是**动态注册**的方式去加载函数的

NATIVE层逆向

3.2 逆向分析

对于静态注册的函数

关注导出表中Java_com_xxxx_xxxx的函数
【修改函数参数和标准类型，便于分析】(快捷键y)

【演示】

在找不到或者无法定位的情况下，考虑动态注册的情况，关注**JNI_OnLoad**函数和**init_array段**

NATIVE层逆向

3.2 逆向分析

题目名称：CrackmeN1

描述：
静态注册的Native函数分析
找到正确输入的flag
Flag格式 flag{xxxxx}

NATIVE层逆向

3.2 逆向分析

题目名称：CrackmeN2

描述：
静态注册的Native函数分析
找到正确输入的flag
Flag格式 flag{xxxxx}

NATIVE层逆向

3.2 逆向分析

对于动态注册的函数

关注JNI_OnLoad函数
同样【修改函数参数和标准类型，便于分析】

【演示】

NATIVE层逆向

3.2 逆向分析

题目名称：CrackmeN3

描述：
动态注册的Native函数分析
找到正确输入的flag
Flag格式 flag{xxxxx}

3.3 真题练习

NATIVE层逆向

3.3 真题练习

题目名称：glass.apk

描述：
【题目出自ciscn2021】
找到正确输入的flag

NATIVE层逆向

3.3 真题练习

题目名称： Strange apk

描述：
选自SCTF2019
找到正确的flag

Hint：真正的代码逻辑在哪里呢？

4. 修改APK调试手段与hook

如何修改apk
如何在设备上进行动态调试

4.1 APK的拆包改包

修改APK&调试手段&hook

4.1 APK的拆包改包

在这部分内容中，我们将会学习如何拆解并修改一个apk。
正如我们在windows系统上修改可执行文件一样，修改apk也有它固定的一些套路，掌握修改apk的方法，可以轻松将apk化为己用，也可以让调试和逆向分析事半功倍。

【APK去广告的演示】
【apk打log的演示+之前没讲的smali语法】

修改APK&调试手段&hook

4.1 APK的拆包改包

Apktool的使用

反编译：

$ apktool d bar.apk

$ apktool decode bar.apk // 效果一样 反编译 bar.apk 并将其解压到 bar 目录

$ apktool d bar.apk o baz 反编译 bar.apk 并将其解压到 baz 目录

修改APK&调试手段&hook

4.1 APK的拆包改包

Apktool的使用

重新打包：

$ apktool b bar –p [输出目录] -o new_bar.apk // 将 bar 目录的资源打包成 new_bar.apk

【红字部分是由于高版本可能报错，加上比较好，随便整个目录就行】

【注意】打包完之后的apk需要重新签名和对齐才能安装

签名使用： uber-apk-signer-1.2.1.jar

修改APK&调试手段&hook

4.1 APK的拆包改包

实验1

修改helloworld.apk中的静态字符串。
步骤：

1. 解包
2. 改res中的字符串
3. 打包
4. 签名
5. Adb安装测试apk

修改APK&调试手段&hook

4.1 APK的拆包改包

实验2

通过logcat，获得helloworld.apk中的flag。
步骤：

1. 解包
2. 改smali代码打log
3. 打包
4. 签名
5. Adb安装测试apk

Smali语法参考：https://www.chieng.cn/post/169.html
【注意】增加参数的时候要更改.locals声明的寄存器数量

4.2 frida环境搭建

修改APK&调试手段&hook

4.2 frida环境搭建

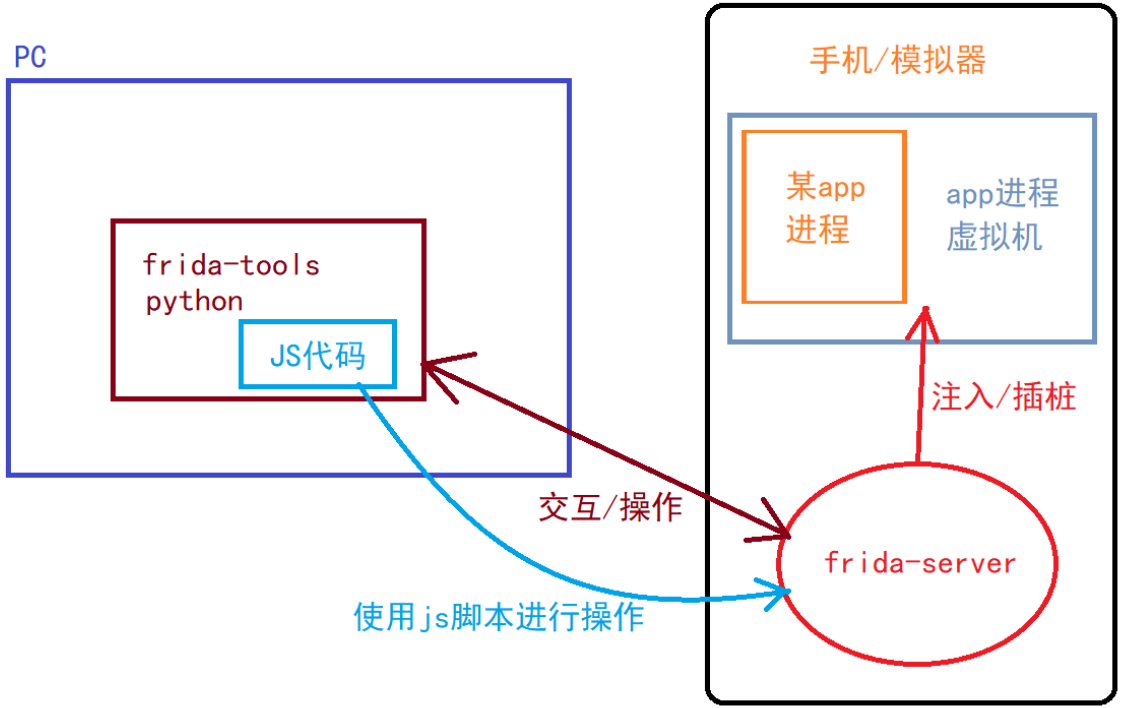
需要的环境和工具：

1个root过的手机/模拟器
Frida-python
Frida-server

修改APK&调试手段&hook

4.2 frida环境搭建

FRIDA原理：



修改APK&调试手段&hook

4.2 frida环境搭建

FRIDA原理：

具体操作app的方式有两种：

1. Spawn（调用）模式【相当于用调试器启动程序】
2. Attach（附加）模式【相当于先启动程序再用调试器附加】

修改APK&调试手段&hook

4.2 frida环境搭建

步骤：

1. Pip install frida-tools

验证：frida –version
显示版本号

修改APK&调试手段&hook

4.2 frida环境搭建

步骤：

2. 下载frida-server

注意版本要和pip安装的版本一致
查看手机架构：getprop ro.product.cpu.abi

下载地址： https://github.com/frida/frida/releases

修改APK&调试手段&hook

4.2 frida环境搭建

步骤：

3. 将frida-server推到模拟器/手机上

adb push C:\Users\thinkpad\Downloads\frida-server-15.1.1-android-x86 /data/local/tmp

Chmod 777 frida-server-15.1.1-android-x86

./ frida-server-15.1.1-android-x86

修改APK&调试手段&hook

4.2 frida环境搭建

步骤：

4. 为模拟器/手机做端口转发，目的是让frida找到设备

adb forward tcp:27042 tcp:27042　　# 把电脑xxxx端口发到手机xxxx端口
adb forward tcp:27043 tcp:27043
查看 Android 进程列表 frida-ps –R

【如果直接用命令行启动则可以不需要这样操作】

修改APK&调试手段&hook

4.2 frida环境搭建

常用frida命令：

查看frida能连接的设备：

frida-ls-devices

访问模拟器：

Frida-ps –U

https://frida.re/docs/frida-ps/

```
# Connect Frida to an iPad over USB and list running processes
$ frida-ps -U

# List running applications
$ frida-ps -Ua

# List installed applications
$ frida-ps -Uai

# Connect Frida to the specific device
$ frida-ps -D 0216027d1d6d3a03
```

```
# 查看当前包名和主Activity
adb shell dumpsys window | findstr mCurrentFocus
```

```
# 跟踪某个函数
frida-trace -U -f Name -i "函数名"
# frida-trace -U -f com.autonavi.minimap -i "getRequestParams"

# 跟踪某个方法
frida-trace -U -f Name -m "方法名"
# frida-trace -U -f com.autonavi.minimap -m "MapLoader"
```

修改APK&调试手段&hook

4.2 frida环境搭建

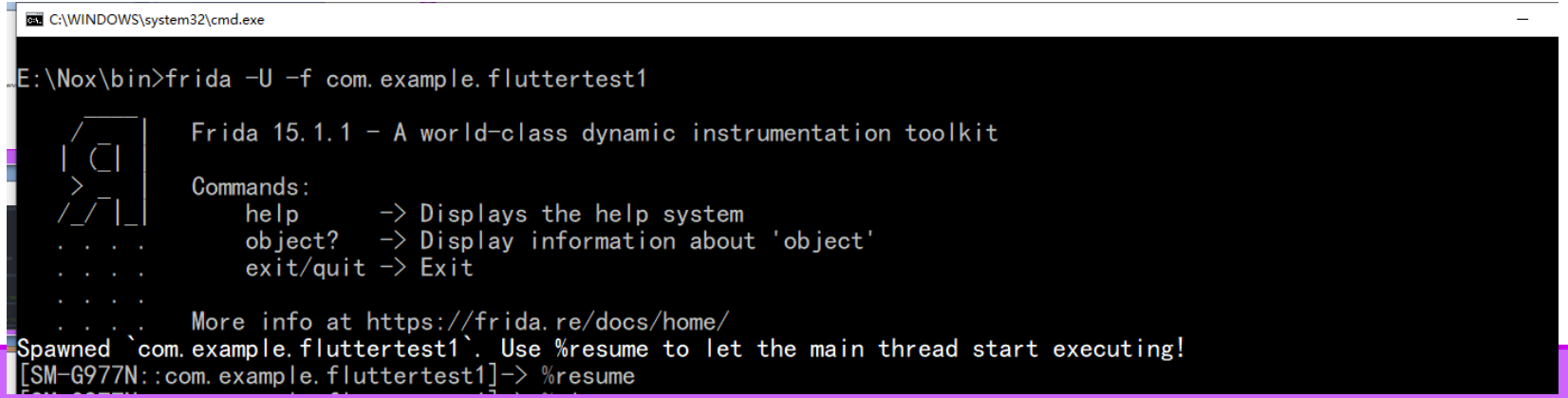
常用frida命令：

命令行启动一个app:

frida –U [包名] / frida –U [包名]

frida –U [包名] -l hook.js # -l表示load a script

frida –U –f [包名] -l hook.js # -f代表spawn模式(打开新app) --no-pause可以不暂停



修改APK&调试手段&hook

4.2 frida环境搭建

常用方法参考文章：https://blog.csdn.net/zhy025907/article/details/89512096

通过python脚本进行hook：

```
import frida  #导入frida模块
import sys    #导入sys模块

# js 代码
jscode = """
Java.perform(function(){
    var Log = Java.use('android.util.Log'); //获得Log类
    send(Log);
    Log.d("Mz1", 'test');  // 直接调用这个函数
    Log.d.overload('java.lang.String', 'java.lang.String').implementation = function(tag, s){
                send(tag + s);
                var result = this.d("Mz1", "hook!");
                return result;
    }
});
"""

def on_message(message,data): #js中执行send函数后要回调的函数
    print(message)

# 得到设备并劫持进程com.mz.helloworld
#（开始用get_usb_device函数用来获取设备，但是一直报错找不到设备，改用get_remote_device函数即可解决这个问题）
device = frida.get_remote_device()

# attach方式启动
# process = device.attach('com.mz.helloworld')  # 这个方法在模拟器上有点问题
# 或者spawn方式启动：
pid = device.spawn(["com.mz.helloworld"])
device.resume(pid)
process = device.attach(pid)

script = process.create_script(jscode) #创建js脚本
script.on('message',on_message) #加载回调函数，也就是js中执行send函数规定要执行的python函数
script.load() #加载脚本
sys.stdin.read()
```

4.3 frida hook实验

修改APK&调试手段&hook

4.3 frida hook实验

实验1：

使用frida hook helloworld.apk
更改logcat输出的内容

修改APK&调试手段&hook

4.3 frida hook实验

实验1：

参考代码：

```
Java.perform(function(){
    var Log = Java.use('android.util.Log'); //获得Log类
    send(Log);
    Log.d("Mz1", 'test');   // 直接调用这个函数
    Log.d.overload('java.lang.String', 'java.lang.String').implementation =
function(tag, s){
                send(tag + s);
                var result = this.d("Mz1", "hook!");
                return result;
    }
});
```

4.4 真题练习

修改APK&调试手段&hook

4.4 真题练习

题目名称： PixelShooter.apk

描述：
选自MRCTF2020
找到正确的flag

Hint： unity逆向

修改APK&调试手段&hook

4.4 真题练习

题目名称： PixelShooter.apk

进一步的要求：

是否能够通过修改apk轻松过关呢？

修改APK&调试手段&hook

4.4 真题练习

题目名称：bang.apk

描述：
选自网鼎杯 2020 青龙组
找到正确的flag

Hint：可能需要脱壳
试试frida-dexdump?

END

感谢观看！