第 5 章

CHAPTER 5

静态逆向工具

5.1 Apktool 工具

视频 8

5.1.1 Apktool 基础与用法

Apktool 是最常用的反编译 Apk 文件的工具，它可以将 Apk 包中的 Dex 文件和资源文件解码，并在修改后重新构建并打包。在 GitHub 上下载它的源码和发布的版本，网址为 https://github.com/iBotPeaches/Apktool，直接下载 Jar 包。

如图 5.1 所示为 Apktool 的下载页面。

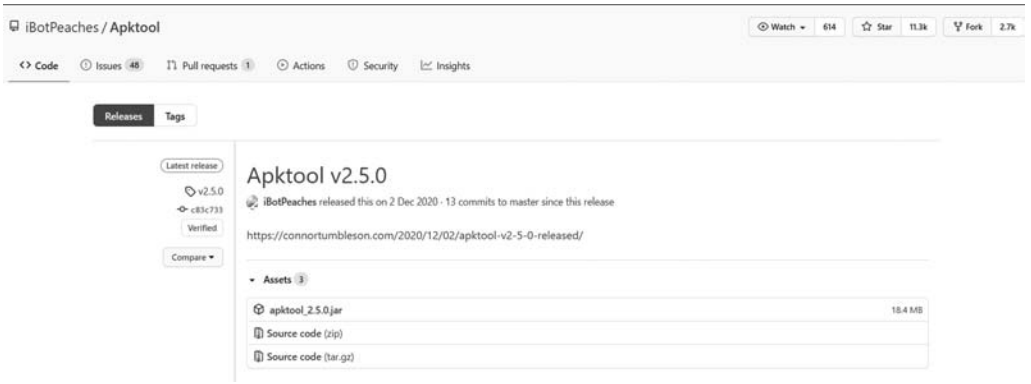

图 5.1 Apktool 的下载页面

下载完毕后在命令行工具中使用 Java 命令执行，查看它的功能与用法。

```
$ java -jar apktool.jar
```

如图 5.2 所示为 Apktool 所使用的参数。

从图 5.2 中可以看到以下参数：

```
usage: apktool
 -advance,--advanced   prints advance information.
 -version,--version    prints the version then exits
usage: apktool if|install-framework [options] <framework.apk>
```

```
node1@node1:~/test_example/tools$ java -jar apktool.jar
Apktool v2.4.1-dirty - a tool for reengineering Android apk files
with smali v2.3 and baksmali v2.3
Copyright 2014 Ryszard Wi膿niewski <brut.alll@gmail.com>
Updated by Connor Tumbleson <connor.tumbleson@gmail.com>

usage: apktool
 -advance,--advanced   prints advance information.
 -version,--version    prints the version then exits
usage: apktool if|install-framework [options] <framework.apk>
 -p,--frame-path <dir>   Stores framework files into <dir>.
 -t,--tag <tag>          Tag frameworks using <tag>.
usage: apktool d[ecode] [options] <file_apk>
 -f,--force              Force delete destination directory.
 -o,--output <dir>       The name of folder that gets written. Default is apk.out
 -p,--frame-path <dir>   Uses framework files located in <dir>.
 -r,--no-res             Do not decode resources.
 -s,--no-src             Do not decode sources.
 -t,--frame-tag <tag>    Uses framework files tagged by <tag>.
usage: apktool b[uild] [options] <app_path>
 -f,--force-all          Skip changes detection and build all files.
 -o,--output <dir>       The name of apk that gets written. Default is dist/name.apk
 -p,--frame-path <dir>   Uses framework files located in <dir>.

For additional info, see: http://ibotpeaches.github.io/Apktool/
For smali/baksmali info, see: https://github.com/JesusFreke/smali
```

图 5.2　Apktool 所使用的参数

```
 -p, --frame-path <dir>   Stores framework files into <dir>.
 -t, --tag <tag>          Tag frameworks using <tag>.
usage: apktool d[ecode] [options] <file_apk>
 -f, --force              Force delete destination directory.
 -o, --output <dir>       The name of folder that gets written. Default is apk.out
 -p, --frame-path <dir>   Uses framework files located in <dir>.
 -r, --no-res             Do not decode resources.
 -s, --no-src             Do not decode sources.
 -t, --frame-tag <tag>    Uses framework files tagged by <tag>.
usage: apktool b[uild] [options] <app_path>
 -f, --force-all          Skip changes detection and build all files.
 -o, --output <dir>       The name of apk that gets written. Default is dist/name.apk
 -p, --frame-path <dir>   Uses framework files located in <dir>.
```

Apktool 主要有两种用法：一个是 d 参数，代表解码 Apk 包；另一个是 b 参数，代表构建 Apk 包，其中 d 参数下有两个选项需要注意。

- -r,--no-res：这个参数指定了在反编译 Apk 的过程中不去处理包内的资源文件，也就是不解码 resource.arsc 文件和 AndroidManifest.xml 文件。Apktool 在处理某些应用的时候可能会因为资源文件解码错误而导致反编译失败。如果只是对源码进行修改可以绕过资源的反编译，只对 Dex 文件进行反编译处理，在二次打包时 Apktool 会将资源文件原封不动地复制回包内。
- -s,--no-src：这个参数指定了在反编译过程中不去处理包内的源码文件，这样包内的 Dex 文件就不会被反编译成 Smali，而 AndroidManifest.xml 与 resource.arsc 会被解码。

这两个参数在实际使用 Apktool 的过程中十分有用。还要注意另一个参数，这个参数在读 Apktool 源码的时候会看到：--force-manifest，虽然这个参数不在 usage 列表中，但是可以使用的，这个参数的作用是无论是否处理资源文件都强制将 AndroidManifest.xml 文件进行解码。

5.1.2　Apktool 源码分析

一般逆向工作中所使用的 Apktool 工具是已经编译好的发布版，但是由于各家的 Apk 包情况不同，部分应用针对 Apktool 工具做了防护，发布版会出现反编译失败的情况。这时可以根据自己的需要动手修改源码，对 Apktool 工具进行自定义。首先从 GitHub 将项目下载或者 clone 下来。

如图 5.3 所示为 Apktool 源码目录。

```
node1@node1:~/test_example/tools/Apktool$ ls -l
total 92
drwxrwxr-x 4 node1 node1  4096 Dec 11 17:19 brut.apktool
drwxrwxr-x 3 node1 node1  4096 Dec 11 17:19 brut.j.common
drwxrwxr-x 3 node1 node1  4096 Dec 11 17:19 brut.j.dir
drwxrwxr-x 3 node1 node1  4096 Dec 11 17:19 brut.j.util
-rw-rw-r-- 1 node1 node1  4418 Dec 11 17:19 build.gradle
-rw-rw-r-- 1 node1 node1   513 Dec 11 17:19 CONTRIBUTORS.md
drwxrwxr-x 3 node1 node1  4096 Dec 11 17:19 gradle
-rwxr-xr-x 1 node1 node1  5960 Dec 11 17:19 gradlew
-rw-rw-r-- 1 node1 node1  2942 Dec 11 17:19 gradlew.bat
-rw-rw-r-- 1 node1 node1 12605 Dec 11 17:19 INTERNAL.md
-rw-rw-r-- 1 node1 node1 11627 Dec 11 17:19 LICENSE
-rw-rw-r-- 1 node1 node1  2587 Dec 11 17:19 README.md
-rw-rw-r-- 1 node1 node1  2597 Dec 11 17:19 ROADMAP.md
drwxrwxr-x 5 node1 node1  4096 Dec 11 17:19 scripts
-rw-rw-r-- 1 node1 node1   240 Dec 11 17:19 SECURITY.md
-rw-rw-r-- 1 node1 node1   140 Dec 11 17:19 settings.gradle
```

图 5.3　Apktool 源码目录

分析 Apktool 源码的出发点在 brut.Apktool 目录。其中的 Apktool-cli 目录下有一个 Main.java 文件，这就是 Apktool 的程序入口。

如图 5.4 所示为 Apktool 的 main()函数。

```
public class Main {
    public static void main(String[] args) throws IOException, InterruptedException, BrutException {

        // headless
        System.setProperty("java.awt.headless", "true");

        // set verbosity default
        Verbosity verbosity = Verbosity.NORMAL;

        // cli parser
        CommandLineParser parser = new DefaultParser();
        CommandLine commandLine;
```

图 5.4　Apktool 的 main()函数

Main.java 文件负责处理运行程序时输入的参数，并根据参数选择对应的功能。

如图 5.5 所示为 Apktool 处理参数的代码逻辑。

使用 Apktool 时添加参数“-d --decode”，程序会调用 cmdDecode 方法进行反编译 Apk 相关操作。在 cmdDecode 方法中会看见一个对象 ApkDecoder，这是 Apktool 负责具体反编译工作的对象，它的定义在 Apktool/brut.apktool/apktool-lib/src/main/java/brut/androlib/目录下的 ApkDecoder.java 文件中。接下来继续深入讨论 ApkDecoder 对象。

如图 5.6 所示为 Apktool 项目的 Androidlib 目录。

下面从 ApkDecoder 类的 decode 方法入手分析 Apktool 的反编译功能。

校验 Apk 文件初始化反编译目录：

```
boolean cmdFound = false;
for (String opt : commandLine.getArgs()) {
    if (opt.equalsIgnoreCase("d") || opt.equalsIgnoreCase("decode")) {
        cmdDecode(commandLine);
        cmdFound = true;
    } else if (opt.equalsIgnoreCase("b") || opt.equalsIgnoreCase("build")) {
        cmdBuild(commandLine);
        cmdFound = true;
    } else if (opt.equalsIgnoreCase("if") || opt.equalsIgnoreCase("install-framework")) {
        cmdInstallFramework(commandLine);
        cmdFound = true;
    } else if (opt.equalsIgnoreCase("empty-framework-dir")) {
        cmdEmptyFrameworkDirectory(commandLine);
        cmdFound = true;
    } else if (opt.equalsIgnoreCase("list-frameworks")) {
        cmdListFrameworks(commandLine);
        cmdFound = true;
    } else if (opt.equalsIgnoreCase("publicize-resources")) {
        cmdPublicizeResources(commandLine);
        cmdFound = true;
    }
}
```

图 5.5　Apktool 处理参数的代码逻辑

```
node1@node1:~/test_example/tools/Apktool/brut.apktool/apktool-lib/src/main/java/brut/androlib$ ls -l
total 84
-rw-rw-r-- 1 node1 node1  1182 Dec 11 17:19 AndrolibException.java
-rw-rw-r-- 1 node1 node1 31833 Dec 11 17:19 Androlib.java
-rw-rw-r-- 1 node1 node1 17462 Dec 11 17:19 ApkDecoder.java
-rw-rw-r-- 1 node1 node1  1547 Dec 11 17:19 ApkOptions.java
-rw-rw-r-- 1 node1 node1  3013 Dec 11 17:19 ApktoolProperties.java
drwxrwxr-x 2 node1 node1  4096 Dec 11 17:19 err
drwxrwxr-x 2 node1 node1  4096 Dec 11 17:19 meta
drwxrwxr-x 2 node1 node1  4096 Dec 11 17:19 mod
drwxrwxr-x 6 node1 node1  4096 Dec 11 17:19 res
drwxrwxr-x 2 node1 node1  4096 Dec 11 17:19 src
```

图 5.6　Apktool 项目的 Androidlib 目录

```
if (!mForceDelete && outDir.exists()) {
    throw new OutDirExistsException();
}

if (!mApkFile.isFile() || !mApkFile.canRead()) {
    throw new InFileNotFoundException();
}

try {
    OS.rmdir(outDir);
} catch (BrutException ex) {
    throw new AndrolibException(ex);
}
outDir.mkdirs();
```

上述代码负责判断输入的 Apk 文件是否有错以及初始化反编译目录。如图 5.7 所示为处理资源文件的代码逻辑。

这段代码先判断 Apk 包中是否有资源文件，也就是 resources.arsc 文件。如果存在，则根据是否使用了参数“-r,--no-res”指定了不解码资源文件，默认是 DECODE_RESOURCES_FULL，也就是解码所有资源文件，如果有 Manifest 文件，则调用 mAndrolib.decodeManifestWithResources()方法解码 AndroidManifest.xml 文件，然后调用 mAndrolib.

```
if (hasResources()) {
    switch (mDecodeResources) {
        case DECODE_RESOURCES_NONE:
            mAndrolib.decodeResourcesRaw(mApkFile, outDir);
            if (mForceDecodeManifest == FORCE_DECODE_MANIFEST_FULL) {
                setTargetSdkVersion();
                setAnalysisMode(mAnalysisMode, true);

                // done after raw decoding of resources because copyToDir overwrites dest files
                if (hasManifest()) {
                    mAndrolib.decodeManifestWithResources(mApkFile, outDir, getResTable());
                }
            }
            break;
        case DECODE_RESOURCES_FULL:
            setTargetSdkVersion();
            setAnalysisMode(mAnalysisMode, true);

            if (hasManifest()) {
                mAndrolib.decodeManifestWithResources(mApkFile, outDir, getResTable());
            }
            mAndrolib.decodeResourcesFull(mApkFile, outDir, getResTable());
            break;
    }
```

图 5.7　处理资源文件的代码逻辑

decodeResourcesFull()方法解码资源文件。如果指定了不解码资源文件，则调用 mAndrolib.decodeResourcesRaw()方法，接着判断是否设置了强制解码 AndroidManifest.xml 文件；如果设置了强制解码，则调用 mAndrolib.decodeManifestWithResources()方法解码文件。

接下来进入 mAndrolib 对象的类 Androlib，看看 Apktool 是怎么具体处理资源文件的。

如图 5.8 所示为 Androlib 类的部分逻辑截图。

```
public class Androlib {
    private final AndrolibResources mAndRes = new AndrolibResources();
    protected final ResUnknownFiles mResUnknownFiles = new ResUnknownFiles();
    public ApkOptions apkOptions;
    private int mMinSdkVersion = 0;

    public Androlib(ApkOptions apkOptions) {
        this.apkOptions = apkOptions;
        mAndRes.apkOptions = apkOptions;
    }

    public Androlib() {
        this.apkOptions = new ApkOptions();
        mAndRes.apkOptions = this.apkOptions;
    }

    public ResTable getResTable(ExtFile apkFile)
            throws AndrolibException {
        return mAndRes.getResTable(apkFile, true);
    }
```

图 5.8　Androlib 类的部分逻辑截图

首先来看 decodeResourcesRaw()方法。

decodeResourcesRaw()方法：

```
public void decodeResourcesRaw(ExtFile apkFile, File outDir)
        throws AndrolibException {
    try {
```

```
            LOGGER.info("Copying raw resources...");
            apkFile.getDirectory().copyToDir(outDir, APK_RESOURCES_FILENAMES);
        } catch (DirectoryException ex) {
            throw new AndrolibException(ex);
        }
    }
```

这个方法将包内的资源文件全部直接复制到输出目录下。也就是说,如果通过参数“-r,--no-res”指定不解码资源文件,则直接将资源文件复制出来。

再来看一下用来解码资源文件的方法 decodeResourcesFull()。

decodeResourcesFull()方法:

```
public void decodeResourcesFull(ExtFile apkFile, File outDir, ResTable resTable)
            throws AndroidException{

    mAndRes.decode(resTable, apkFile, outDir);

}
```

decodeResourcesFull()方法调用 mAndRes 对象的 decode()方法,对资源文件进行解码:

```
public void decode(ResTable resTable, ExtFile ApkFile, File outDir)
          throws AndrolibException {
   Duo<ResFileDecoder, AXmlResourceParser> duo = getResFileDecoder();
   ResFileDecoder fileDecoder = duo.m1;
   ResAttrDecoder attrDecoder = duo.m2.getAttrDecoder();

   attrDecoder.setCurrentPackage(resTable.listMainPackages().iterator().next());
   Directory inApk, in = null, out;

   try {
      out = new FileDirectory(outDir);
      inApk = ApkFile.getDirectory();
      out = out.createDir("res");
      if (inApk.containsDir("res")) {
         in = inApk.getDir("res");
      }
      if (in == null && inApk.containsDir("r")) {
         in = inApk.getDir("r");
      }
      if (in == null && inApk.containsDir("R")) {
         in = inApk.getDir("R");
      }
   } catch (DirectoryException ex) {
      throw new AndrolibException(ex);
   }

   ExtMXSerializer xmlSerializer = getResXmlSerializer();
   for (ResPackage pkg : resTable.listMainPackages()) {
```

```
        attrDecoder.setCurrentPackage(pkg);

        LOGGER.info("Decoding file-resources...");
        for (ResResource res : pkg.listFiles()) {
            fileDecoder.decode(res, in, out);
        }

        LOGGER.info("Decoding values */* XMLs...");
        for (ResValuesFile valuesFile : pkg.listValuesFiles()) {
            generateValuesFile(valuesFile, out, xmlSerializer);
        }
        generatePublicXml(pkg, out, xmlSerializer);
    }

    AndrolibException decodeError = duo.m2.getFirstError();
    if (decodeError != null) {
        throw decodeError;
    }
}
```

接着来分析负责处理 AndroidManifest.xml 文件的方法 mAndRes.decodeManifest-WithResources()：

```
public void decodeManifestWithResources(ExtFile apkFile, File outDir, ResTable resTable)
        throws AndrolibException {
    mAndRes.decodeManifestWithResources(resTable, apkFile, outDir);
}
```

decodeManifestWithResources()同样调用了 mAndRes 对象中的同名方法，来解析 AndroidManifest.xml：

```
public void decodeManifestWithResources(ResTable resTable, ExtFile apkFile, File outDir)
        throws AndrolibException {
    Duo<ResFileDecoder, AXmlResourceParser> duo = getResFileDecoder();
    ResFileDecoder fileDecoder = duo.m1;
    ResAttrDecoder attrDecoder = duo.m2.getAttrDecoder();
    attrDecoder.setCurrentPackage(resTable.listMainPackages().iterator().next());
    Directory inApk, in = null, out;
    try {
        inApk = apkFile.getDirectory();
        out = new FileDirectory(outDir);
        LOGGER.info("Decoding AndroidManifest.xml with resources...");
        fileDecoder.decodeManifest(inApk, "AndroidManifest.xml", out, "AndroidManifest.xml");
        if (!resTable.getAnalysisMode()) {
            adjustPackageManifest(resTable, outDir.getAbsolutePath() + File.separator +
"AndroidManifest.xml");
            ResXmlPatcher.removeManifestVersions(new File(
                    outDir.getAbsolutePath() + File.separator + "AndroidManifest.xml"));
```

```
            mPackageId = String.valueOf(resTable.getPackageId());
        }
    } catch (DirectoryException ex) {
        throw new AndrolibException(ex);
    }
}
```

如果 Apk 中不存在 resources.arsc 文件,则不参照属性的引用对 AndroidManifest 文件解码:

```
else {
    // if there's no resources.arsc, decode the manifest without looking
    // up attribute references
    if (hasManifest()) {
        if (mDecodeResources == DECODE_RESOURCES_FULL
            || mForceDecodeManifest == FORCE_DECODE_MANIFEST_FULL) {
            mAndrolib.decodeManifestFull(mApkFile, outDir, getResTable());
        }
        else {
            mAndrolib.decodeManifestRaw(mApkFile, outDir);
        }
    }
}
```

资源文件处理完毕后再处理源码文件。一个 Apk 中可能有多个 Dex 文件,所以 decode()方法先处理第一个 Dex 文件 classes.dex,然后再根据是否存在其他的 Dex 文件进行下一步。

如图 5.9 所示为 Apktool 判断是否存在其他 Dex 文件的代码。

```
if (hasMultipleSources()) {
    // foreach unknown dex file in root, lets disassemble it
    Set<String> files = mApkFile.getDirectory().getFiles( recursive: true);
    for (String file : files) {
        if (file.endsWith(".dex")) {
            if (!file.equalsIgnoreCase( anotherString: "classes.dex")) {
                switch (mDecodeSources) {
                case DECODE_SOURCES_NONE:
                    mAndrolib.decodeSourcesRaw(mApkFile, outDir, file);
                    break;
                case DECODE_SOURCES_SMALI:
                    mAndrolib.decodeSourcesSmali(mApkFile, outDir, file, mBakDeb, mApi);
                    break;
                }
            }
        }
    }
}
```

图 5.9 判断是否存在其他 Dex 文件

针对每个 Dex 文件检查传入的参数,如果参数指定了不处理 Dex 文件,则调用 decodeSourcesRaw()方法,直接将 Dex 文件复制到目标目录下。

```
public void decodeSourcesRaw(ExtFile apkFile, File outDir, String filename)
          throws AndrolibException {
   try {
      LOGGER.info("Copying raw " + filename + " file...");
      apkFile.getDirectory().copyToDir(outDir, filename);
   } catch (DirectoryException ex) {
      throw new AndrolibException(ex);
   }
}
```

如果是默认情况，也就是需要解码 Dex 文件，则会调用 decodeSourcesSmali()方法：

```
public void decodeSourcesSmali(File apkFile, File outDir, String filename, boolean bakdeb, int
api)
          throws AndrolibException {
   try {
      File smaliDir;
      if (filename.equalsIgnoreCase("classes.dex")) {
         smaliDir = new File(outDir, SMALI_DIRNAME);
      } else {
         smaliDir = new File(outDir, SMALI_DIRNAME + "_" + filename.substring(0, filename.
indexOf(".")));
      }
      OS.rmdir(smaliDir);
      smaliDir.mkdirs();
      LOGGER.info("Baksmaling " + filename + "...");
      SmaliDecoder.decode(apkFile, smaliDir, filename, bakdeb, api);
   } catch (BrutException ex) {
      throw new AndrolibException(ex);
   }
}
```

decodeSourcesSmali()方法调用 Baksmali 组件将 Dex 文件反编译成 Smali 文件。

5.2　JEB 工具

JEB 是一款强大的跨平台的 Android 静态分析工具，提供了类似于 IDA Pro 的方法交叉引用与重命名功能，同时提供脚本化功能，用于自动化分析和对抗代码混淆。

5.2.1　JEB 安装

从 JEB 官网 https://www.pnfsoftware.com/jeb/下载软件包并解压。如图 5.10 所示为 JEB 程序目录。

JEB 的运行需要依赖 JDK8 或以上的 JDK 环境。提前下载安装 JDK 并设置系统变量 JAVA_HOME。JEB 提供了可在 Windows、Linux、macOS 上运行的 UI 客户端，在相应的

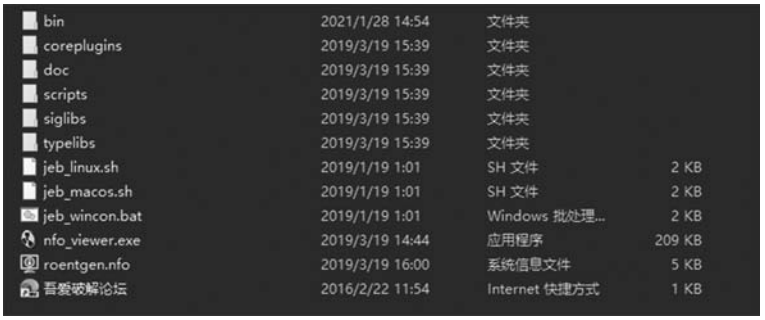

图 5.10 JEB 程序目录

系统执行对应文件：Windows 执行 jeb_wincon. bat；Linux 执行 jeb_linux. sh；macOS 执行 jeb_macos. sh。

如图 5.11 所示为 JEB 运行起来的效果。

图 5.11 JEB 运行起来的效果

5.2.2 JEB 静态分析

接下来通过使用 JEB 分析一个应用来熟悉它的用法。本书将从 JEB 官网 https://www. pnfsoftware. com/jeb/manual/下载官方实例应用 Raasta. Apk 作为此章节的测试应用。

如图 5.12 所示为下载 Raasta. Apk 的页面。

从文件菜单中打开 Raasta. Apk 文件，JEB 会使用这个 Apk 文件创建一个新的项目，并通过以下几个 Android 分析组件去处理它：

- Apk 组件负责拆解应用文件，解码它的 AndroidManifest 文件和资源文件。
- Dex 组件负责解析应用包中的 Dex 字节码文件。
- Xml 组件负责处理 Android 应用资源目录下的 XML 资源文件。

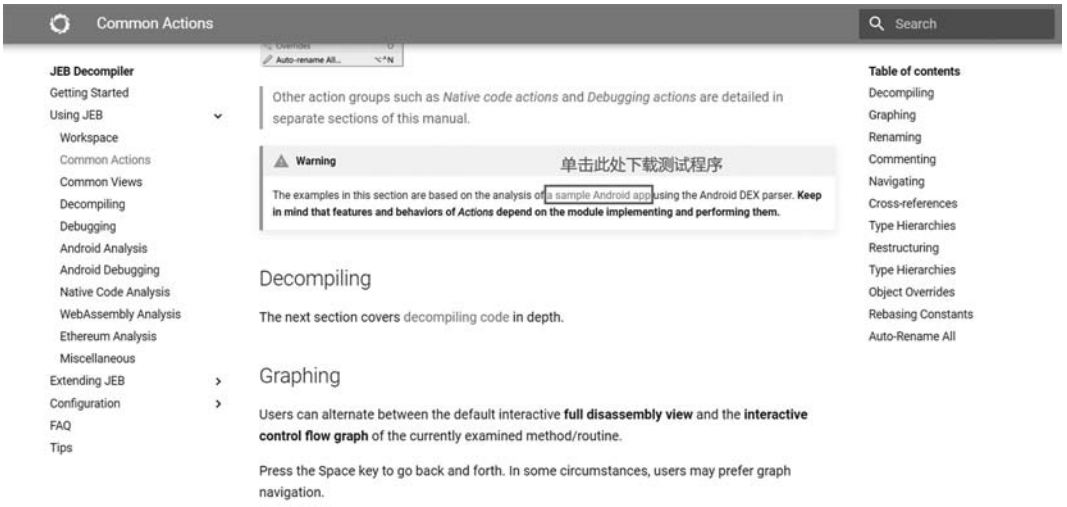

图 5.12　下载 Raasta.Apk 的页面

• 签名组件负责分析应用的签名。

打开项目后，左侧的项目浏览树会展示 Apk 包内的各类文件和目录，左侧下方的 Bytecode 结构树会显示 Dex 字节码反编译出来的源码项目结构。主要的 Dex 窗口是在项目分析完毕后自动打开的，展示 Dex 文件反编译出来的 Smali 语句。

如图 5.13 所示为 JEB 的主界面截图。

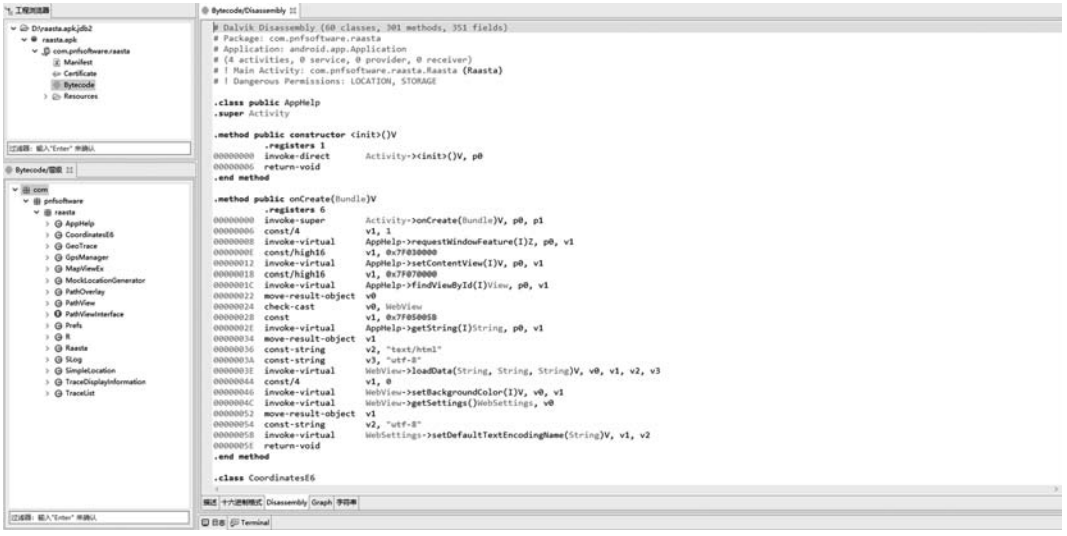

图 5.13　JEB 的主界面截图

接下来介绍 JEB 在进行静态分析时的常用功能。

1. 反编译功能

在主界面的 Dex 字节码窗口，滚动到需要反编译的部分区域，按 Tab 键就可以将 Smali 源码反编译成 Java 源码。这时会打开另一个窗口，里面是选择区域所属的 Java 类的 Java 源码。

如图 5.14 所示为 JEB 反编译应用的效果。

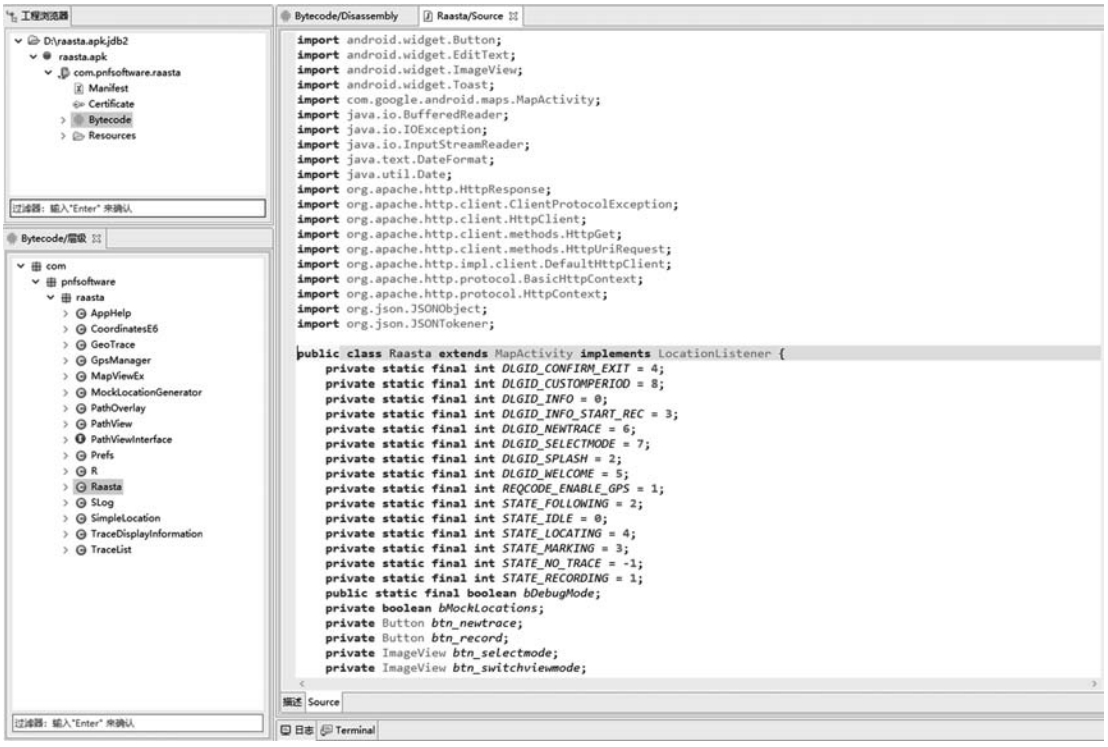

图 5.14　JEB 反编译应用的效果

在 Java 源码界面再按 Tab 键就会回到对应的 Smali 语句。

2. 重命名

静态分析时,可能会遇到代码被混淆的情况,多个方法名和变量被转化成类似于 a()、ab()之类毫无意义的形式。为了应对这种情况,一个比较重要的需求是可以对代码中的各项,比如类型、方法、例程、类变量、数据项或者包名进行重命名。JEB 提供了这一功能。

- 定位并单击需要重命名的项目。
- 按 N 键或者选择 Action 栏中的 Rename 选项。
- 输入新的名称。

如图 5.15 所示为 JEB 对反编译代码的项目进行重命名操作。

在“重命名”窗口中按 Ctrl+空格键可以查看之前的重命名历史记录。

3. 添加注释

在代码中的任意位置,按“/”键打开添加注释界面,输入注释内容。注释会附加在所选择的语句的后面。

如图 5.16 所示为 JEB 为代码添加注释。

4. 导航

在做静态分析时经常需要去找某个调用方法或结构体的定义。在 JEB 中,单击选中项目并按回车键或者在项目上双击,就会跳转到显示该项定义的窗口。可以使用快捷键“Alt+左箭头”或“Alt+右箭头”进行向前或者向后导航。

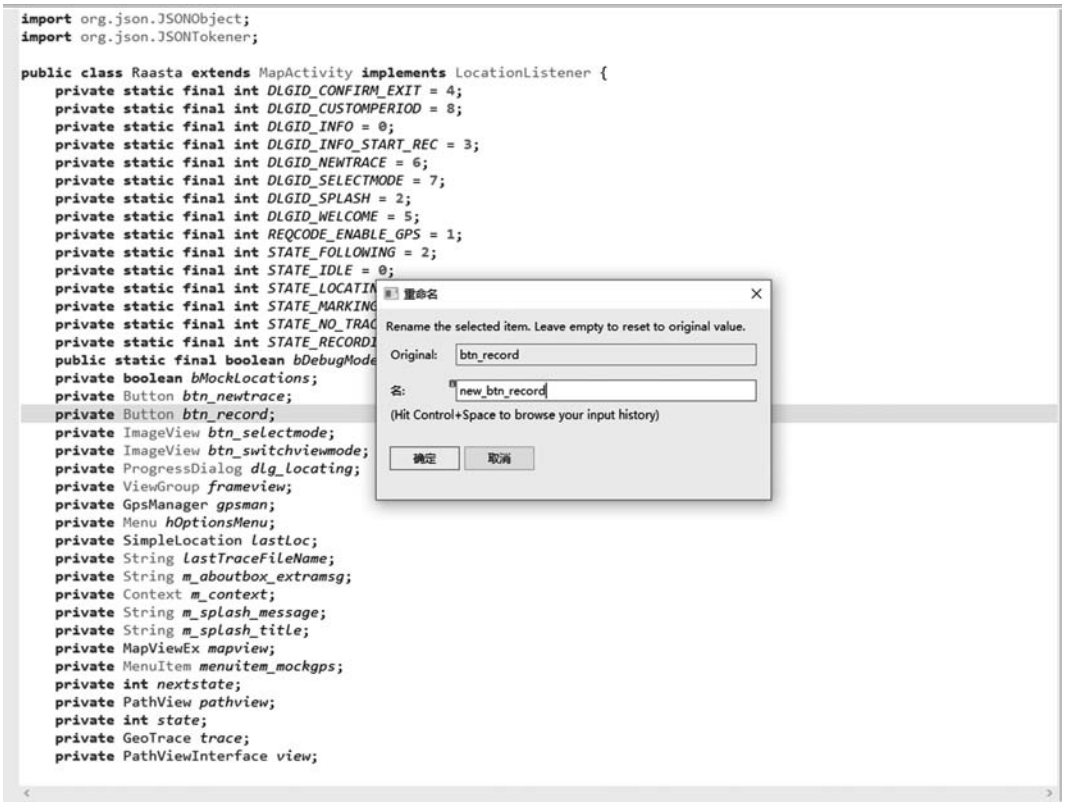

图 5.15　JEB 对反编译代码的项目进行重命名操作

图 5.16　JEB 为代码添加注释

如图 5.17 所示为选择 set_lastSplashSequence()函数调用。

```
Prefs.set_lastSplashSequence(((Context)this), v4);
this.m_splash_title = v2.optString("cap");
this.m_splash_message = v2.optString("msg");
if(this.m_splash_title.length() <= 0) {
    return true;
}
```

图 5.17　选择 set_lastSplashSequence()函数调用

如图 5.18 所示为 JEB 跳转到 set_lastSplashSequence()函数的定义。

```
public static void set_lastSplashSequence(Context arg4, int arg5) {
    SharedPreferences$Editor v0 = arg4.getSharedPreferences("global", 0).edit();
    v0.putInt("splashSeq", arg5);
    v0.commit();
}
```

图 5.18　JEB 跳转到 set_lastSplashSequence()函数的定义

5. 交叉引用

除了查看某个引用项的定义,有时逆向人员还需要查看某个引用项在整个项目中的引用情况。在 JEB 中选择某项,按 X 键打开交叉引用窗口,双击窗口中的项目跳转到引用点。

如图 5.19 所示为查看函数的交叉引用。

图 5.19　查看函数的交叉引用

6. 重构

JEB 提供了强大的重构能力,允许使用者在项目中创建新的包并把某个包内的类移动到新的或者是已存在的其他包中。

作为示范,此处将使用这个功能创建一个新包 com. newPack,然后将 com. pnfsoftware. raasta 包内的 AppHelp 类移动到新包 com. newPack 中。首先按 K 键新建一个包 com. newPack。

如图 5.20 所示为在 JEB 中新建一个包。

图 5.20　在 JEB 中新建一个包

然后按 L 键将 AppHelp 类移动到 com.newPack 包中。

如图 5.21 所示为在 JEB 中移动类。

图 5.21　在 JEB 中移动类

7. 修改常量

这个功能允许选择整数常量以哪种进制形式显示。选择常量后，按 B 键循环选择插件提供的进制。通常插件提供八进制、十进制、十六进制方式，其他的插件可能会额外提供二进制方式，或者是非常规的显示方式，比如基于字符。

如图 5.22 所示为以十进制显示选中的整数常量。

```
if(arg6 > 0) {
    int v0 = Prefs.get_traceGpsPeriod(((Context)this), this.lastTraceFileName);
    if(v0 <= 0) {
        v0 = Prefs.get_GpsPeriod(((Context)this));
    }

    this.gpsman.request_updates(((long)(v0 * 1000)), 0f);
}
```

图 5.22　以十进制显示选中的整数常量

如图 5.23 所示为以八进制显示选中的整数常量。

```
if(arg6 > 0) {
    int v0 = Prefs.get_traceGpsPeriod(((Context)this), this.lastTraceFileName);
    if(v0 <= 0) {
        v0 = Prefs.get_GpsPeriod(((Context)this));
    }

    this.gpsman.request_updates(((long)(v0 * 01750)), 0f);
}
```

图 5.23　以八进制显示选中的整数常量

如图 5.24 所示为以十六进制显示选中的整数常量。

```
if(arg6 > 0) {
    int v0 = Prefs.get_traceGpsPeriod(((Context)this), this.lastTraceFileName);
    if(v0 <= 0) {
        v0 = Prefs.get_GpsPeriod(((Context)this));
    }

    this.gpsman.request_updates(((long)(v0 * 0x3E8)), 0f);
}
```

图 5.24　以十六进制显示选中的整数常量

视频 9

5.3　Jadx-gui 工具

Jadx 是一个将 Dex 字节码文件反编译成 Java 的开源工具，作者同时提供了 UI 客户端方便进行动态调试。Github 地址为 https://github.com/skylot/jadx。下载 Jar 包到本地后运行 Java 命令启动：

```
java - jar jadx - gui.jar
```

如图 5.25 所示为 Jadx-gui 的启动界面。

图 5.25　Jadx-gui 的启动界面

Jadx-gui 的主要特性是：

- 处理 Apk、Dex、Aar 和 Zip 文件，将其中的 Dalvik 字节码反编译成 Java 类。
- 反编译 AndroidManifest.xml 和解码其他的资源文件和 resources.arsc 文件。
- 添加了反混淆功能。
- 使用高亮语法显示反编译出来的代码。

如图 5.26 所示为 Jadx-gui 反编译代码的截图。

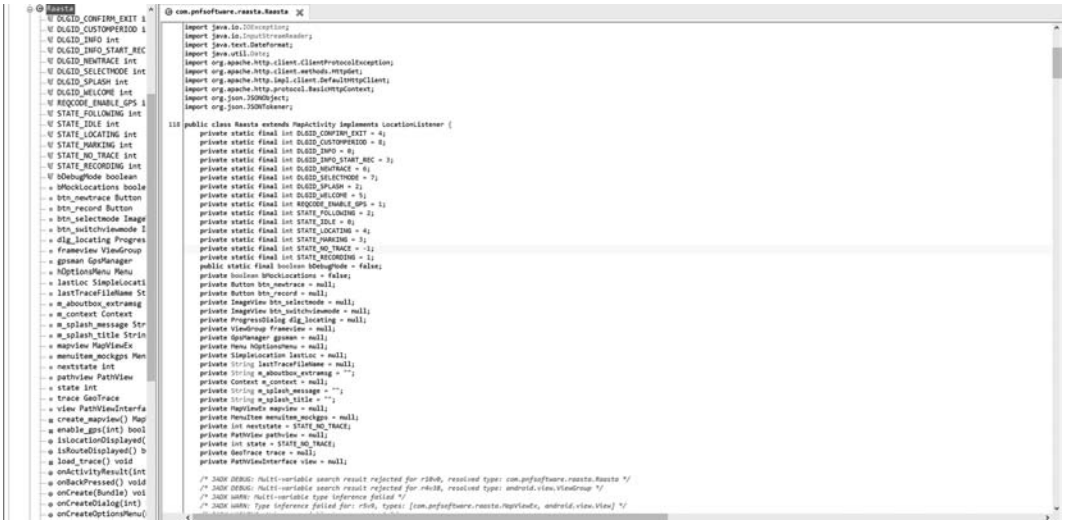

图 5.26　Jadx-gui 反编译代码的截图

• 跳转到项目的定义。

在类、方法和类变量项目上按 Ctrl＋鼠标左键跳转到该项目的定义。如图 5.27 所示为选择 set_lastTraceFileName()函数。

```
public void onClick(DialogInterface dlg, int id) {
    String name = e.getText().toString();
    String filename = GeoTrace.generate_timebased_filename();
    GeoTrace t = new GeoTrace(Raasta.this.m_context, filename, true);
    t.set_name(name);
    t.save();
    Prefs.set_lastTraceFileName(Raasta.this.m_context, filename);
    Prefs.set_traceGpsPeriod(Raasta.this.m_context, filename, Prefs.get_GpsPeriod(Raasta.this.m_context));
    Raasta.this.btn_newtrace.setVisibility(Raasta.DLGID_CUSTOMPERIOD);
    Raasta.this.enable_gps(0);
    Raasta.this.load_trace();
}
```

图 5.27　选择 set_lastTraceFileName()函数

如图 5.28 所示为跳转到 set_lastTraceFileName()函数的定义。

```
public static void set_lastTraceFileName(Context context, String s) {
    SharedPreferences.Editor ed = context.getSharedPreferences("global", 0).edit();
    ed.putString("lastTraceFileName", s);
    ed.commit();
}
```

图 5.28　跳转到 set_lastTraceFileName()函数的定义

在 AndroidManifest. xml 中可以直接跳转到类的定义，如图 5.29 所示为在 AndroidManifest. xml 中找到 TraceList 类。

```
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
    <activity android:label="@string/prefs_title" android:name=".Prefs"/>
    <activity android:label="@string/trlist_title" android:name=".TraceList"/>
    <activity android:theme="@style/Theme.Dialog" android:label="@string/help_title" android:name=".AppHelp"/>
</application>
```

图 5.29　在 AndroidManifest. xml 中找到 TraceList 类

如图 5.30 所示为从 AndroidManifest 中跳转到 TraceList 类。

```
package com.pnfsoftware.raasta;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.app.ListActivity;
import android.app.ProgressDialog;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.LinearLayout;
import android.widget.ListView;
import android.widget.Toast;
import com.pnfsoftware.raasta.GeoTrace;
import java.io.File;
import java.util.Vector;

72 public class TraceList extends ListActivity implements View.OnClickListener, DialogInterface.OnClickListener {
    private static final int DLGID_CHOOSE = 1;
    private static final int DLGID_CREATE = 0;
    private static final int DLGID_DELETE = 5;
    private static final int DLGID_EXPORT = 3;
    private static final int DLGID_RENAME = 2;
    private static final int DLGID_REQCLOSE = 6;
    private static final int ID_CHOOSE_DELETE = 8195;
    private static final int ID_CHOOSE_EXPORT = 8194;
    private static final int ID_CHOOSE_OPEN = 8192;
    private static final int ID_CHOOSE_RENAME = 8193;
    private static final int ID_CREATE_TRACENAME = 4096;
    private static final int ID_RENAME_TRACENAME = 4097;
    private static int[] action_btn_ids = {ID_CHOOSE_OPEN, ID_CHOOSE_RENAME, ID_CHOOSE_EXPORT, ID_CHOOSE_DELETE};
    private static String[] m_action_names;
    private Activity m_activity;
    private Context m_context;
    private Dialog m_dlg_choose = null;
    private Dialog m_dlg_create = null;
    private Dialog m_dlg_delete = null;
    private Dialog m_dlg_export = null;
    private Dialog m_dlg_exporting = null;
    private Dialog m_dlg_rename = null;
    private Dialog m_dlg_reqclose = null;
    private String m_path_exportedTrace = null;
    private String m_selected_filename = "";
    private String m_selected_name = "";
    private Thread m_thread_exporting = null;
    private Vector<String> m_trfilenames;
    private Vector<String> m_trnames;
```

图 5.30　从 AndroidManifest 中跳转到 TraceList 类

• 查找项目的引用。

在项目上右击,在弹出的快捷菜单中选择 Find Usage 命令,就会弹出该项目的所有引用。如图 5.31 所示为在 Jadx-gui 中查找引用。

图 5.31　在 Jadx-gui 中查找引用

• 文本搜索。

Jadx-gui 提供全局文本搜索，在整个项目的范围内查找字符串，可以匹配类名、方法名、变量名、代码。

如图 5.32 所示为在 Jadx-gui 中进行文本搜索。

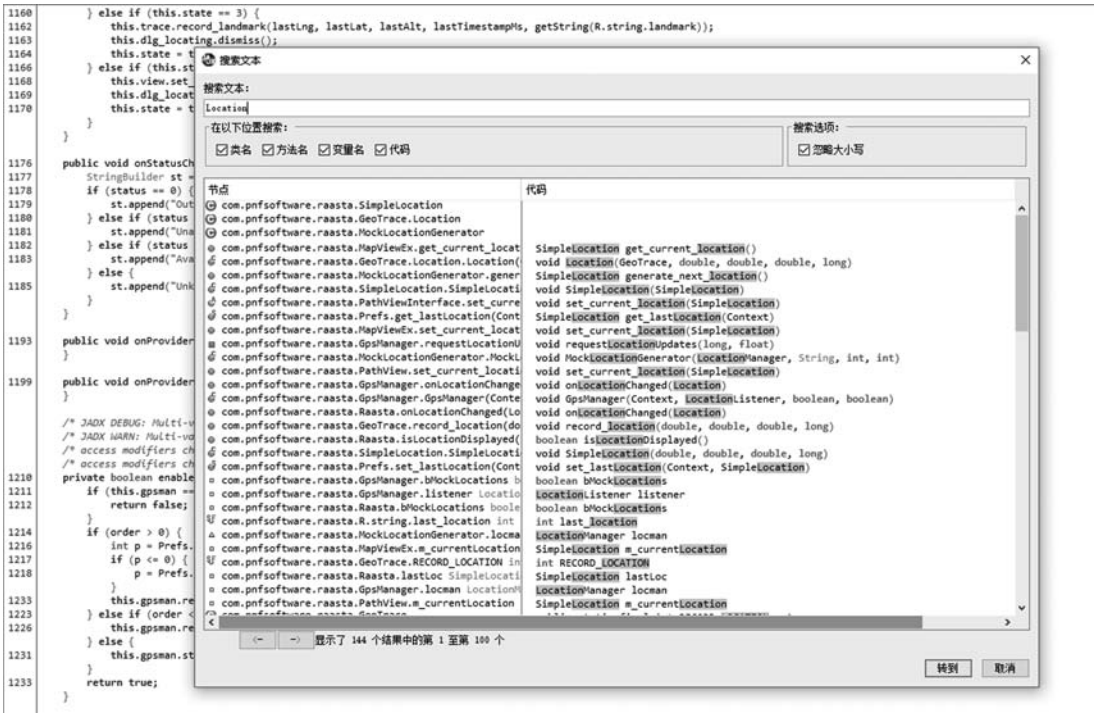

图 5.32　在 Jadx-gui 中进行文本搜索

Jadx-gui 功能相比较于 JEB 要少一些，但是 Jadx-gui 是开源项目，轻量小巧，足够应付静态分析的任务。

5.4　010-editor 工具

010-editor 可以说是目前最强大的一款十六进制编辑器，可以编辑与查看各种十六进制文件，可以通过加载不同的文件模板解析各种文件格式。

5.4.1　010-editor 解析 so 文件

010-editor 官网有 elf 的文件模板网址为 https://www.sweetscape.com/010editor/repository/files/ELF.bt，将其复制下来，单击 Templates 选项，选择 New Templates 打开模板。使用 010-editor 打开 so 文件后按 F5 键即可使用对应的模板。

010-editor 模板对照 elf 文件定义的格式从十六进制中直接解析出 so 文件的各段表。010-editor 的优点是可以更加直观地去定位某个结构的值在整个文件中的位置，可以配合

Hex editor 等十六进制文件编辑器对 so 文件直接进行修改。

如图 5.33 所示为使用 010-editor 打开 so 文件的界面。

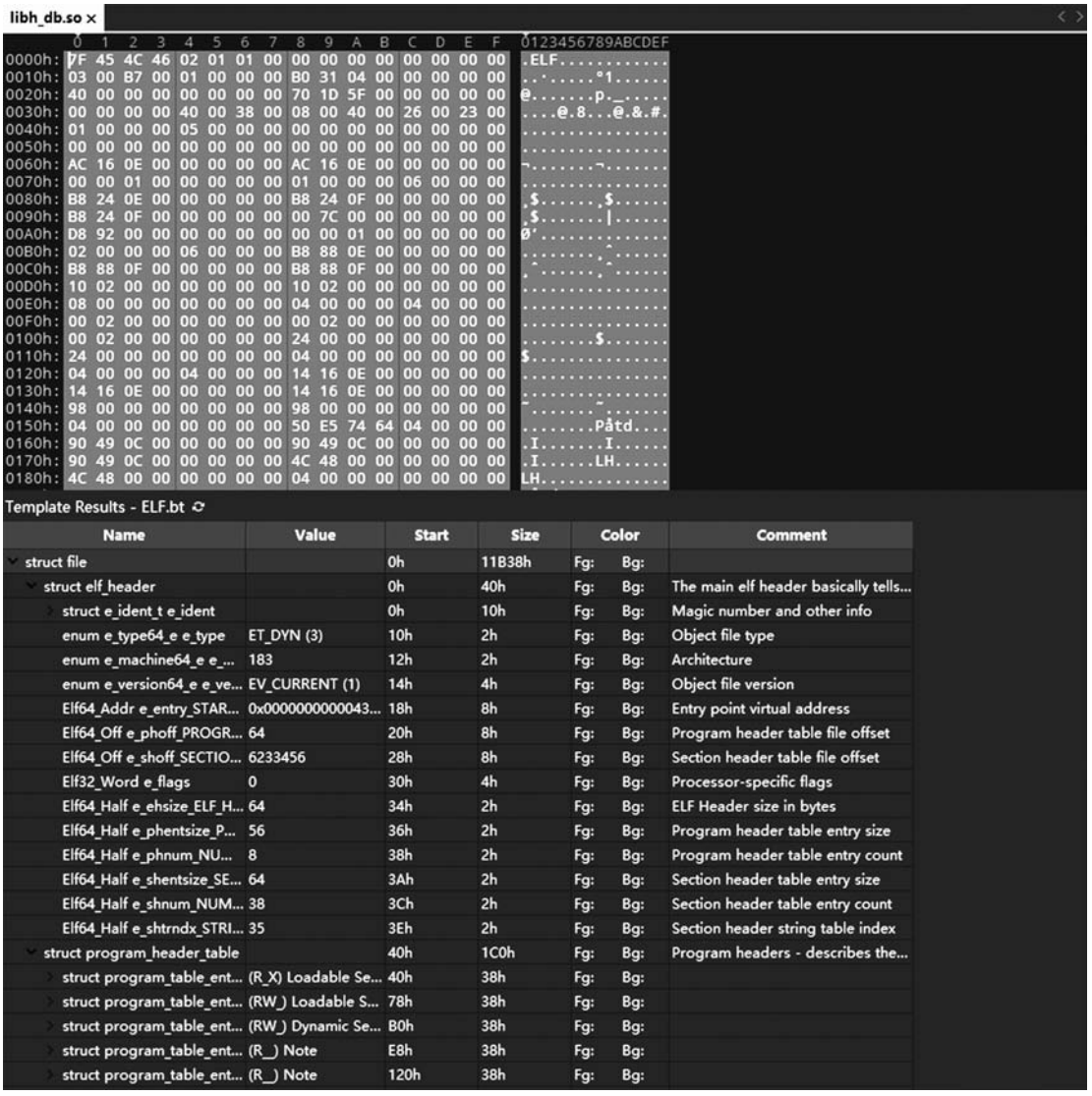

图 5.33　使用 010-editor 打开 so 文件的界面

5.4.2　010-editor 解析 Dex 文件

Dex 的文件模板可以在 010-editor 官网上找到，网址为 https://www.sweetscape.com/010editor/repository/files/DEX.bt，按照上面的做法加载模板，打开 Dex 文件。可以将 Apk 包用 Zip 工具解压缩获得 Dex 文件。

如图 5.34 所示为使用 010-editor 打开 Dex 文件的界面。

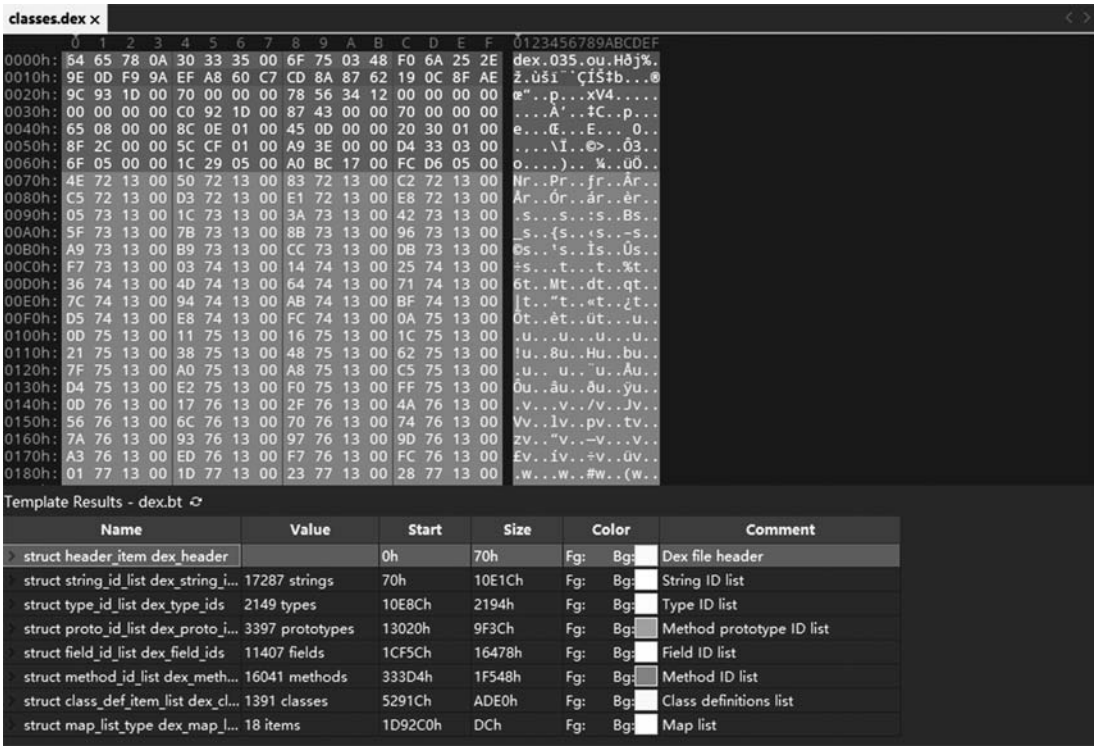

图 5.34　使用 010-editor 打开 Dex 文件的界面

5.5　本章小结

本章介绍了对 Android 应用进行静态分析常用的工具以及使用方法，包括各种工具对 Native 层与 Java 层的静态逆向分析。互联网中还有许多集成了图形化操作界面的 Android 分析工具，比如 Android Killer，这些分析工具本质上是对 Baksmali、Apktool 等工具的集成。这些工具还会在后面实战篇中陆续出现，本书也会结合具体需求介绍更多的用于逆向分析的实战工具。