Android安全攻防战，反编译与混淆技术完全解析（下1）

2016-03-28 安卓应用频道

(点击上方公众号，可快速关注)

> 来源：郭霖
> 链接：http://blog.csdn.net/guolin_blog/article/details/50451259

在上一篇文章当中，我们学习了Android程序反编译方面的知识，包括反编译代码、反编译资源、以及重新打包等内容。通过这些内容我们也能看出来，其实我们的程序并没有那么的安全。可能资源被反编译影响还不是很大，重新打包又由于有签名的保护导致很难被盗版，但代码被反编译就有可能会泄漏核心技术了，因此一款安全性高的程序最起码要做到的一件事就是：对代码进行混淆。

混淆代码并不是让代码无法被反编译，而是将代码中的类、方法、变量等信息进行重命名，把它们改成一些毫无意义的名字。因为对于我们而言可能Cellphone类的call()方法意味着很多信息，而A类的b()方法则没有任何意义，但是对于计算机而言，它们都是平等的，计算机不会试图去理解Cellphone是什么意思，它只会按照设定好的逻辑来去执行这些代码。所以说混淆代码可以在不影响程序正常运行的前提下让破解者很头疼，从而大大提升了程序的安全性。

今天是我们Android安全攻防战系列的下篇，本篇文章的内容建立在上篇的基础之上，还没有阅读过的朋友可以先去参考 Android安全攻防战，反编译与混淆技术完全解析（上）。

混淆

本篇文章中介绍的混淆技术都是基于Android Studio的，Eclipse的用法也基本类似，但是就不再为Eclipse专门做讲解了。

我们要建立一个Android Studio项目，并在项目中添加一些能够帮助我们理解混淆知识的代码。这里我准备好了一些，我们将它们添加到Android Studio当中。

首先新建一个MyFragment类，代码如下所示：

```
public class MyFragment extends Fragment {

    private String toastTip = "toast in MyFragment";
```

```
    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInsta
nceState) {
        View view = inflater.inflate(R.layout.fragment_layout, container, false);
        methodWithGlobalVariable();
        methodWithLocalVariable();
        return view;
    }

    public void methodWithGlobalVariable() {
        Toast.makeText(getActivity(), toastTip, Toast.LENGTH_SHORT).show();
    }

    public void methodWithLocalVariable() {
        String logMessage = "log in MyFragment";
        logMessage = logMessage.toLowerCase();
        System.out.println(logMessage);
    }

}
```

可以看到，MyFragment是继承自Fragment的，并且MyFragment中有一个全局变量。onCreateView()方法是Fragment的生命周期函数，这个不用多说，在onCreateView()方法中又调用了methodWithGlobalVariable()和methodWithLocalVariable()方法，这两个方法的内部分别引用了一个全局变量和一个局部变量。

接下来新建一个Utils类，代码如下所示：

```
public class Utils {

    public void methodNormal() {
        String logMessage = "this is normal method";
        logMessage = logMessage.toLowerCase();
        System.out.println(logMessage);
    }

    public void methodUnused() {
```

```
        String logMessage = "this is unused method";
        logMessage = logMessage.toLowerCase();
        System.out.println(logMessage);
    }

}
```

这是一个非常普通的工具类，没有任何继承关系。Utils中有两个方法methodNormal()和methodUnused()，它们的内部逻辑都是一样的，唯一的据别是稍后methodNormal()方法会被调用，而methodUnused()方法不会被调用。

下面再新建一个NativeUtils类，代码如下所示：

```
public class NativeUtils {

    public static native void methodNative();

    public static void methodNotNative() {
        String logMessage = "this is not native method";
        logMessage = logMessage.toLowerCase();
        System.out.println(logMessage);
    }

}
```

这个类中同样有两个方法，一个是native方法，一个是非native方法。

最后，修改MainActivity中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    private String toastTip = "toast in MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        getSupportFragmentManager().beginTransaction().add(R.id.fragment, new MyFragment()).commit();
        Button button = (Button) findViewById(R.id.button);
```

```
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                methodWithGlobalVariable();
                methodWithLocalVariable();
                Utils utils = new Utils();
                utils.methodNormal();
                NativeUtils.methodNative();
                NativeUtils.methodNotNative();
                Connector.getDatabase();
            }
        });
    }

    public void methodWithGlobalVariable() {
        Toast.makeText(MainActivity.this, toastTip, Toast.LENGTH_SHORT).show();
    }

    public void methodWithLocalVariable() {
        String logMessage = "log in MainActivity";
        logMessage = logMessage.toLowerCase();
        System.out.println(logMessage);
    }

}
```

可以看到，MainActivity和MyFragment类似，也是定义了methodWithGlobalVariable()和methodWithLocalVariable()这两个方法，然后MainActivity对MyFragment进行了添加，并在Button的点击事件里面调用了自身的、Utils的、以及NativeUtils中的方法。注意调用native方法需要有相应的so库实现，不然的话就会报UnsatisefiedLinkError，不过这里其实我也并没有真正的so库实现，只是演示一下让大家看看混淆结果。点击事件的最后一行调用的是LitePal中的方法，因为我们还要测试一下引用第三方Jar包的场景，到LitePal项目的主页去下载最新的Jar包，然后放到libs目录下即可。

完整的build.gradle内容如下所示：

```
apply plugin: 'com.android.application'

android {
```

```
    compileSdkVersion 23
    buildToolsVersion "23.0.2"

    defaultConfig {
        applicationId "com.example.guolin.androidtest"
        minSdkVersion 15
        targetSdkVersion 23
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:23.2.0'
}
```

好的，到这里准备工作就已经基本完成了，接下来我们就开始对代码进行混淆吧。

混淆APK

在Android Studio当中混淆APK实在是太简单了，借助SDK中自带的Proguard工具，只需要修改build.gradle中的一行配置即可。可以看到，现在build.gradle中minifyEnabled的值是false，这里我们只需要把值改成true，打出来的APK包就会是混淆过的了。如下所示：

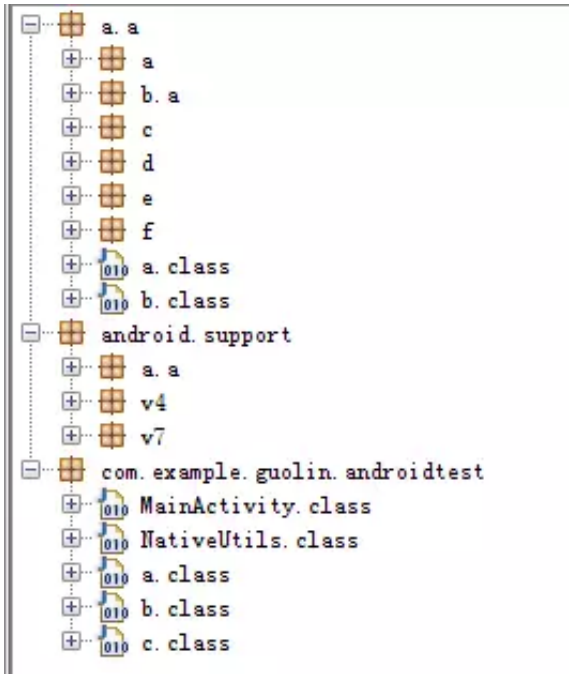
```
release {
    minifyEnabled true
    proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
}
```

其中minifyEnabled用于设置是否启用混淆，proguardFiles用于选定混淆配置文件。注意这里是在release闭包内进行配置的，因此只有打出正式版的APK才会进行混淆，Debug版的APK是不会混淆的。当然这也是非常合理的，因为Debug版的APK文件我们只会用来内部测

试，不用担心被人破解。

那么现在我们来打一个正式版的APK文件，在Android Studio导航栏中点击Build->Generate Signed APK，然后选择签名文件并输入密码，如果没有签名文件就创建一个，最终点击Finish完成打包，生成的APK文件会自动存放在app目录下。除此之外也可以在build.gradle文件当中添加签名文件配置，然后通过gradlew assembleRelease来打出一个正式版的APK文件，这种方式APK文件会自动存放在app/build/outputs/apk目录下。

那么现在已经得到了APK文件，接下来就用上篇文章中学到的反编译知识来对这个文件进行反编译吧，结果如下图所示：



很明显可以看出，我们的代码混淆功能已经生效了。

下面我们尝试来阅读一下这个混淆过后的代码，最顶层的包名结构主要分为三部分，第一个a.a已经被混淆的面目全非了，但是可以猜测出这个包下是LitePal的所有代码。第二个android.support可以猜测出是我们引用的android support库的代码，第三个com.example.guolin.androidtest则很明显就是我们项目的主包名了，下面将里面所有的类一个个打开看一下。

首先MainActivity中的代码如下所示：

```
package com.example.guolin.androidtest;

import android.os.Bundle;
import android.support.v4.a.ad;
import android.support.v4.a.aq;
import android.support.v7.a.u;
import android.widget.Button;
import android.widget.Toast;
import java.io.PrintStream;

public class MainActivity
  extends u
{
  private String l = "toast in MainActivity";

  public void k()
  {
    Toast.makeText(this, this.l, 0).show();
  }

  public void l()
  {
    String str = "log in MainActivity".toLowerCase();
    System.out.println(str);
  }

  protected void onCreate(Bundle paramBundle)
  {
    super.onCreate(paramBundle);
    setContentView(2130968601);
    f().a().a(2131492944, new b()).a();
    ((Button)findViewById(2131492945)).setOnClickListener(new a(this));
  }
}
```

可以看到，MainActivity的类名是没有混淆的，onCreate()方法也没有被混淆，但是我们定义的方法、全局变量、局部变量都被混淆了。

再来打开下一个类NativeUtils，如下所示：

```
package com.example.guolin.androidtest;

import java.io.PrintStream;

public class NativeUtils
{
  public static void a()
  {
    String str = "this is not native method".toLowerCase();
    System.out.println(str);
  }

  public static native void methodNative();
}
```

NativeUtils的类名没有被混淆，其中声明成native的方法也没有被混淆，但是非native方法的方法名和局部变量都被混淆了。

接下来是a类的代码，如下所示：

```
package com.example.guolin.androidtest;

import android.view.View;
import android.view.View.OnClickListener;

class a
  implements View.OnClickListener
{
  a(MainActivity paramMainActivity) {}

  public void onClick(View paramView)
  {
    this.a.k();
    this.a.l();
    new c().a();
    NativeUtils.methodNative();
    NativeUtils.a();
    a.a.e.c.b();
  }
}
```

很明显，这个是MainActivity中按钮点击事件的匿名类，在onClick()方法中的调用代码虽然都被混淆了，但是调用顺序是不会改变的，对照源代码就可以看出哪一行是调用的什么方法了。

再接下来是b类，代码如下所示：

```
package com.example.guolin.androidtest;

import android.os.Bundle;
import android.support.v4.a.t;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Toast;
import java.io.PrintStream;

public class b
  extends t
{
  private String Z = "toast in MyFragment";

  public void H()
  {
    Toast.makeText(b(), this.Z, 0).show();
  }

  public void I()
  {
    String str = "log in MyFragment".toLowerCase();
    System.out.println(str);
  }

  public View a(LayoutInflater paramLayoutInflater, ViewGroup paramViewGroup, Bundle paramBundle)
  {
    paramLayoutInflater = paramLayoutInflater.inflate(2130968602, paramViewGroup, false);
    H();
    I();
    return paramLayoutInflater;
  }
}
```

虽然被混淆的很严重，但是我们还是可以看出这个是MyFragment类。其中所有的方法名、全局变量、局部变量都被混淆了。

最后再来看下c类，代码如下所示：

```
package com.example.guolin.androidtest;

import java.io.PrintStream;

public class c
{
  public void a()
  {
    String str = "this is normal method".toLowerCase();
    System.out.println(str);
  }
}
```

c类中只有一个a方法，从字符串的内容我们可以看出，这个是Utils类中的methodNormal()方法。

我为什么要创建这样的一个项目呢？因为从这几个类当中很能看出一些问题，接下来我们就分析一下上面的混淆结果。

首先像Utils这样的普通类肯定是会被混淆的，不管是类名、方法名还是变量都不会放过。除了混淆之外Utils类还说明了一个问题，就是minifyEnabled会对资源进行压缩，因为Utils类中我们明明定义了两个方法，但是反编译之后就只剩一个方法了，因为另外一个方法没有被调用，所以认为是多余的代码，在打包的时候就给移除掉了。不仅仅是代码，没有被调用的资源同样也会被移除掉，因此minifyEnabled除了混淆代码之外，还可以起到压缩APK包的作用。

接着看一下MyFragment，这个类也是混淆的比较彻底的，基本没有任何保留。那有些朋友可能会有疑问，Fragment怎么说也算是系统组件吧，就算普通方法名被混淆了，至少像onCreateView()这样的生命周期方法不应该被混淆吧？其实生命周期方法会不会被混淆和我们使用Fragment的方式有关，比如在本项目中，我使用的是android.support.v4.app.Fragment，support-v4包下的，就连Fragment的源码都被一起混淆了，因此生命周期方法当然也不例外了。但如果你使用的是android.app.Fragment，这就是调用手机系统中预编译好的代码了，很明显我们的混淆无法影响到系统内置的代码，因此这种情况下onCreateView()方法名就不会被混淆，但其它的方法以及变量仍然会被混淆。

接下来看一下MainActivity，同样也是系统组件之一，但MainActivity的保留程度就比MyFragment好多了，至少像类名、生命周期方法名都没有被混淆，这是为什么呢？根据我亲身测试得出结论，凡是需要在AndroidManifest.xml中去注册的所有类的类名以及从父类重写的方法名都自动不会被混淆。因此，除了Activity之外，这份规则同样也适用于Service、BroadcastReceiver和ContentProvider。

最后看一下NativeUtils类，这个类的类名也没有被混淆，这是由于它有一个声明成native的方法。只要一个类中有存在native方法，它的类名就不会被混淆，native方法的方法名也不会被混淆，因为C++代码要通过包名+类名+方法名来进行交互。 但是类中的别的代码还是会被混淆的。

除此之外，第三方的Jar包都是会被混淆的，LitePal不管是包名还是类名还是方法名都被完完全全混淆掉了。

这些就是Android Studio打正式APK时默认的混淆规则。

那么这些混淆规则是在哪里定义的呢？其实就是刚才在build.gradle的release闭包下配置的proguard-android.txt文件，这个文件存放于<Android SDK>/tools/proguard目录下，我们打开来看一下：

```
# This is a configuration file for ProGuard.
# http://proguard.sourceforge.net/index.html#manual/usage.html
```

```
-dontusemixedcaseclassnames
-dontskipnonpubliclibraryclasses
-verbose

# Optimization is turned off by default. Dex does not like code run
# through the ProGuard optimize and preverify steps (and performs some
# of these optimizations on its own).
-dontoptimize
-dontpreverify
# Note that if you want to enable optimization, you cannot just
# include optimization flags in your own project configuration file;
# instead you will need to point to the
# "proguard-android-optimize.txt" file instead of this one from your
# project.properties file.

-keepattributes *Annotation*
-keep public class com.google.vending.licensing.ILicensingService
-keep public class com.android.vending.licensing.ILicensingService

# For native methods, see http://proguard.sourceforge.net/manual/examples.html#native
-keepclasseswithmembernames class * {
    native <methods>;
}

# keep setters in Views so that animations can still work.
# see http://proguard.sourceforge.net/manual/examples.html#beans
-keepclassmembers public class * extends android.view.View {
   void set*(***);
   *** get*();
}

# We want to keep methods in Activity that could be used in the XML attribute onClick
-keepclassmembers class * extends android.app.Activity {
   public void *(android.view.View);
}

# For enumeration classes, see http://proguard.sourceforge.net/manual/examples.html#enumerations
-keepclassmembers enum * {
```

```
    public static **[] values();
    public static ** valueOf(java.lang.String);
}

-keepclassmembers class * implements android.os.Parcelable {
  public static final android.os.Parcelable$Creator CREATOR;
}

-keepclassmembers class **.R$* {
    public static <fields>;
}

# The support library contains references to newer platform versions.
# Dont warn about those in case this app is linking against an older
# platform version.  We know about them, and they are safe.
-dontwarn android.support.**
```

这个就是默认的混淆配置文件了，我们来一起逐行阅读一下。

-dontusemixedcaseclassnames 表示混淆时不使用大小写混合类名。

-dontskipnonpubliclibraryclasses 表示不跳过library中的非public的类。

-verbose 表示打印混淆的详细信息。

-dontoptimize 表示不进行优化，建议使用此选项，因为根据proguard-android-optimize.txt中的描述，优化可能会造成一些潜在风险，不能保证在所有版本的Dalvik上都正常运行。

-dontpreverify 表示不进行预校验。这个预校验是作用在Java平台上的，Android平台上不需要这项功能，去掉之后还可以加快混淆速度。

-keepattributes *Annotation* 表示对注解中的参数进行保留。

```
-keep public class com.google.vending.licensing.ILicensingService
-keep public class com.android.vending.licensing.ILicensingService
```

表示不混淆上述声明的两个类，这两个类我们基本也用不上，是接入Google原生的一些服务时使用的。

```
-keepclasseswithmembernames class * {
    native <methods>;
}
```

表示不混淆任何包含native方法的类的类名以及native方法名，这个和我们刚才验证的结果是一致的。

```
-keepclassmembers public class * extends android.view.View {
   void set*(***);
   *** get*();
}
```

表示不混淆任何一个View中的setXxx()和getXxx()方法，因为属性动画需要有相应的setter和getter的方法实现，混淆了就无法工作了。

```
-keepclassmembers class * extends android.app.Activity {
   public void *(android.view.View);
}
```

表示不混淆Activity中参数是View的方法，因为有这样一种用法，在XML中配置android:onClick=”buttonClick”属性，当用户点击该按钮时就会调用Activity中的buttonClick(View view)方法，如果这个方法被混淆的话就找不到了。

```
-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}
```

表示不混淆枚举中的values()和valueOf()方法，枚举我用的非常少，这个就不评论了。

```
-keepclassmembers class * implements android.os.Parcelable {
  public static final android.os.Parcelable$Creator CREATOR;
}
```

表示不混淆Parcelable实现类中的CREATOR字段，毫无疑问，CREATOR字段是绝对不能改变的，包括大小写都不能变，不然整个Parcelable工作机制都会失败。

```
-keepclassmembers class **.R$* {
    public static <fields>;
```

```
}
```

表示不混淆R文件中的所有静态字段，我们都知道R文件是通过字段来记录每个资源的id的，字段名要是被混淆了，id也就找不着了。

-dontwarn android.support.** 表示对android.support包下的代码不警告，因为support包中有很多代码都是在高版本中使用的，如果我们的项目指定的版本比较低在打包时就会给予警告。不过support包中所有的代码都在版本兼容性上做足了判断，因此不用担心代码会出问题，所以直接忽略警告就可以了。

好了，这就是proguard-android.txt文件中所有默认的配置，而我们混淆代码也是按照这些配置的规则来进行混淆的。经过我上面的讲解之后，相信大家对这些配置的内容基本都能理解了。不过proguard语法中还真有几处非常难理解的地方，我自己也是研究了好久才搞明白，下面和大家分享一下这些难懂的语法部分。

proguard中一共有三组六个keep关键字，很多人搞不清楚它们的区别，这里我们通过一个表格来直观地看下：

关键字	描述
keep	保留类和类中的成员，防止它们被混淆或移除。
keepnames	保留类和类中的成员，防止它们被混淆，但当成员没有被引用时会被移除。
keepclassmembers	只保留类中的成员，防止它们被混淆或移除。
keepclassmembernames	只保留类中的成员，防止它们被混淆，但当成员没有被引用时会被移除。
keepclasseswithmembers	保留类和类中的成员，防止它们被混淆或移除，前提是指名的类中的成员必须存在，如果不存在则还是会混淆。
keepclasseswithmembernames	保留类和类中的成员，防止它们被混淆，但当成员没有被引用时会被移除，前提是指名的类中的成员必须存在，如果不存在则还是会混淆。

除此之外，proguard中的通配符也比较让人难懂，proguard-android.txt中就使用到了很多通配符，我们来看一下它们之间的区别：

通配	描述

符	
<field>	匹配类中的所有字段
<method>	匹配类中的所有方法
<init>	匹配类中的所有构造函数
*	匹配任意长度字符，但不含包名分隔符(.)。比如说我们的完整类名是com.example.test.MyActivity，使用com.*，或者com.exmaple.*都是无法匹配的，因为*无法匹配包名中的分隔符，正确的匹配方式是com.exmaple.*.*，或者com.exmaple.test.*，这些都是可以的。但如果你不写任何其它内容，只有一个*，那就表示匹配所有的东西。
**	匹配任意长度字符，并且包含包名分隔符(.)。比如proguard-android.txt中使用的-dontwarn android.support.**就可以匹配android.support包下的所有内容，包括任意长度的子包。
***	匹配任意参数类型。比如void set*(***)就能匹配任意传入的参数类型，*** get*()就能匹配任意返回值的类型。
…	匹配任意长度的任意类型参数。比如void test(…)就能匹配任意void test(String a)或者是void test(int a, String b)这些方法。

虽说上面表格已经解释的很详细了，但是很多人对于keep和keepclasseswithmembers这两个关键字的区别还是搞不懂。确实，它们之间用法有点太像了，我做了很多次试验它们的结果都是相同的。其实唯一的区别就在于类中声明的成员存不存在，我们还是通过一个例子来直接地看一下，先看keepclasseswithmember关键字：

```
-keepclasseswithmember class * {
    native <methods>;
}
```

这段代码的意思其实很明显，就是保留所有含有native方法的类的类名和native方法名，而如果某个类中没有含有native方法，那就还是会被混淆。

但是如果改成keep关键字，结果会完全不一样：

```
-keep class * {
    native <methods>;
}
```

使用keep关键字后，你会发现代码中所有类的类名都不会被混淆了，因为keep关键字看到class *就认为应该将所有类名进行保留，而不会关心该类中是否含有native方法。当然这样写只会保证类名不会被混淆，类中的成员还是会被混淆的。

比较难懂的用法大概就这些吧，掌握了这些内容之后我们就能继续前进了。

回到Android Studio项目当中，刚才打出的APK虽然已经成功混淆了，但是混淆的规则都是按照proguard-android.txt中默认的规则来的，当然我们也可以修改proguard-android.txt中的规则，但是直接在proguard-android.txt中修改会对我们本机上所有项目的混淆规则都生效，那么有没有什么办法只针对当前项目的混淆规则做修改呢？当然是有办法的了，你会发现任何一个Android Studio项目在app模块目录下都有一个proguard-rules.pro文件，这个文件就是用于让我们编写只适用于当前项目的混淆规则的，那么接下来我们就利用刚才学到的所有知识来对混淆规则做修改吧。

这里我们先列出来要实现的目标：

- 对MyFragment类进行完全保留，不混淆其类名、方法名、以及变量名。
- 对Utils类中的未调用方法进行保留，防止其被移除掉。
- 对第三方库进行保留，不混淆android-support库，以及LitePal库中的代码。

接下文