Android免Root环境下Hook框架Legend原理分析

0x1 应用场景

现如今，免Root环境下的逆向分析已经成为一种潮流！

在2015年之前的iOS软件逆向工程领域，要想对iOS平台上的软件进行逆向工程分析，越狱iOS设备与安装Cydia是必须的！几乎绝大多数的逆向相关的动态调试工具、Hook注入框架都依赖于获取IOS设备的最高访问权限，就技术本身上而言，对当前程序进行Hook与动态调试，只需要拥有与当前程序相同的权限即可，理论上无需对设备进行Root越狱，实际上，在2015年就出现了在非越狱设备上进行插件开发的实用案例，2016年的iOS软件逆向工程界，更是一发不可收拾，各种名越狱环境下的逆向工具与逆向技巧被安全人员所发掘，在没有越狱的iOS设备上进行软件的动态调试与逆向工程已经是主流的趋势胃。这样的情况下，最直接的影响是安全研究人员不再对iOS设备越狱有着强烈的追求了，越狱需求的下降可能会直接影响到iOS设备越狱工具的发布与技术的更新迭代。

同样的，在Android设备的免Root环境下，进行软件动态调试与逆向工程分析的需求更加强烈。免Root环境下动态调试与逆向工程就技术本质而言是可行的，安全研究人员的智慧更是有力的证明了这一点，LBE发布免Root环境下APK双开工具平行空间就是最好的例子，它是打破逆向工程技术的原始格局的第一个大锤！随后的，各种APK多开框架、免Root环境下的Hook、免Root环境下的动态调试等技术都被研究人员公开，这是Android软件逆向工程界的福音，逆向工程人员在以后的逆向分析过程中，可能再也不需要为自己的手机能否越狱而感到苦恼，手上在吃灰淘汰的Android小米机可能就是你的逆向必备工具之一。

好了，说了这么多，无非是告诉大家，开发技术在更新迭代，软件的逆向工程技术也在不停的更新，各位研究软件安全的朋友们，你们跟上了时代的脚步吗？！

0x2 Legend框架简介

`Legend`是Lody开源的一个Android免Root环境下的一个APK Hook框架，代码放在github上：https://github.com/asLody/legend。 该框架代码设计简洁，通用性高，适合逆向工程时一些Hook场景。

先来看看如何使用它。框架提供了两种使用方法：基于`Annotation`注解与代码直接调用。基于`Annotation`注解的Hook技术不是第一次被发现了，在Java开发的世界里，这种技术被广泛使用，大名鼎鼎的基于AOP开发的`Aspectj`就大量使用这种技术。使用`Annotation`方式编写的Java代码有着很强的灵活与扩展性。`Legend`中`Annotation`方式的Hook这样使用：

```
@Hook("android.app.Activity::startActivity@android.content.Intent")
public static void Activity_startActivity(Activity thiz, Intent intent) {
  if (!ALLOW_LAUNCH_ACTIVITY) {
    Toast.makeText(thiz, "I am sorry to turn your Activity down :)",
Toast.LENGTH_SHORT).show();
  } else {
    HookManager.getDefault().callSuper(thiz, intent);
  }
}
```

`@Hook("xxx")`部分指明需要Hook的类与方法以及方法的签名，此处的`Activity_startActivity()`是自己实现的替换`android.app.Activity::startActivity()`的方法，`HookManager.getDefault().callSuper(thiz, intent);`调用是调用原方法。

这种方式Hook的方法，需要执行一次Hook应用操作来激活所有注解Hook，方法是执行下面的方法，传入的YourClass.class是包含了注解的类：

```
HookManager.getDefault().applyHooks(YourClass.class);
```

另一种代码方式进行Hook使用起来更简单，Hook操作只需要一行代码：

```
HookManager.getDefault().hookMethod(originMethod, hookMethod);
```

这是`Legend`提供的demo展示的一个完整实例：

```
package com.legend.demo;

import android.app.Activity;
import android.app.Application;
import android.content.Context;
import android.content.Intent;
import android.telephony.TelephonyManager;
import android.widget.Toast;

import com.lody.legend.Hook;
import com.lody.legend.HookManager;

/**
 * @author Lody
 * @version 1.0
 */
public class App extends Application {

    public static boolean ENABLE_TOAST = true;
    public static boolean ALLOW_LAUNCH_ACTIVITY = true;

    @Override
```

```
    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        HookManager.getDefault().applyHooks(App.class);

    }

    @Hook("android.app.Application::onCreate")
    public static void Application_onCreate(Application app) {
        Toast.makeText(app, "Application => onCreate()",
Toast.LENGTH_SHORT).show();
        HookManager.getDefault().callSuper(app);
    }


    @Hook("android.telephony.TelephonyManager::getSimSerialNumber")
    public static String TelephonyManager_getSimSerialNumber(TelephonyManager
thiz) {
         return "110";
    }


    @Hook("android.widget.Toast::show")
    public static void Toast_show(Toast toast) {
        if (ENABLE_TOAST) {
            HookManager.getDefault().callSuper(toast);
        }
    }

    @Hook("android.app.Activity::startActivity@android.content.Intent")
    public static void Activity_startActivity(Activity activity, Intent
intent) {
        if (!ALLOW_LAUNCH_ACTIVITY) {
            Toast.makeText(activity, "I am sorry to turn your Activity down
:)", Toast.LENGTH_SHORT).show();
        }else {
            HookManager.getDefault().callSuper(activity, intent);
        }
    }

}
```

0x3 原理分析

先来看看Hook注解的实现：

```
// Legend/legendCore/src/main/java/com/lody/legend/Hook.java

package com.lody.legend;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Lody
 * @version 1.0
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Hook {
    String value() default "";
}
```

`@Target(ElementType.METHOD)`指明Hook注解用于修饰类中的`Method`，与之类似的还有`@Target(ElementType.FIELD)`用来修饰类中的`Field`。如果想让注解同时修饰类的`Field`与`Method`，可以这么写：

```
@Target({ElementType.FIELD, ElementType.METHOD})
public @interface Hook{}
    ......
```

`@Retention(RetentionPolicy.RUNTIME)`指明Hook注解以何种形式进行保留。`RetentionPolicy`是一个enum类型，声明如下：

```
public enum RetentionPolicy {
   SOURCE,
   CLASS,
   RUNTIME
  }
```

`SOURCE`表明该注解类型的信息只保留在程序源码里，源码经过编译之后，注解的数据就会消失;`CLASS`表明注解类型的信息除了保留在程序源码里，同时也保留在编译好的class文件里面，但在执行的时候，并不会把这些信息加载到内存中去；`RUNTIME`是最大范围的保留，表示同时在源码与编译好的class文件中保留信息，并且在执行的时候会把这些信息加载到内存中去。

定义好了Hook注解，看它是如何使用的，这就是`HookManager.getDefault().applyHooks()`方法要做的工作，它的代码如下：

```
// Legend/legendCore/src/main/java/com/lody/legend/HookManager.java
public void applyHooks(Class<?> holdClass) {
    for (Method hookMethod : holdClass.getDeclaredMethods()) {
```

```
        Hook hook = hookMethod.getAnnotation(Hook.class);
        if (hook != null) {
            String statement = hook.value();
            String[] splitValues = statement.split("::");
            if (splitValues.length == 2) {
                String className = splitValues[0];
                String[] methodNameWithSignature = splitValues[1].split("@");
                if (methodNameWithSignature.length <= 2) {
                    String methodName = methodNameWithSignature[0];
                    String signature = methodNameWithSignature.length == 2 ?
methodNameWithSignature[1] : "";
                    String[] paramList = signature.split("#");
                    if (paramList[0].equals("")) {
                        paramList = new String[0];
                    }
                    try {
                        Class<?> clazz = Class.forName(className);
                        boolean isResolve = false;
                        for (Method method : clazz.getDeclaredMethods()) {
                            if (method.getName().equals(methodName)) {
                                Class<?>[] types =
method.getParameterTypes();
                                if (paramList.length == types.length) {
                                    boolean isMatch = true;
                                    for (int N = 0; N < types.length; N++) {
                                        if
(!types[N].getName().equals(paramList[N])) {
                                            isMatch = false;
                                            break;
                                        }
                                    }
                                    if (isMatch) {
                                        hookMethod(method, hookMethod);
                                        isResolve = true;
                                        Logger.d("[+++] %s have hooked.",
method.getName());
                                    }
                                }
                            }
                            if (isResolve) {
                                break;
                            }
                        }
                        if (!isResolve) {
                            Logger.e("[---] Cannot resolve Method : %s.",
Arrays.toString(methodNameWithSignature));
                        }
                    } catch (Throwable e) {
                        Logger.e("[---] Error to Load Hook Method From : %s."
```

```
, hookMethod.getName());
                        e.printStackTrace();
                    }

                }else {
                    Logger.e("[---] Can't split method and signature : %s.",
Arrays.toString(methodNameWithSignature));
                }
            }else {
                Logger.e("[---] Can't understand your statement : [%s].",
statement);
            }
        }
    }
}
```

该方法遍历类的所有方法，查找匹配注解信息中指定的方法，方法是：对于需要Hook的`Class`类`holdClass`，调用它的`getDeclaredMethods()`获取所有声明的方法，依次调用每个类的方法的`getAnnotation()`获取注解信息，取到的注解信息保存在`String`类型的`statement`变量中，类与完整的方法签名以“::”进行分隔，方法签名中的方法名与参数签名使用“@”进行分隔，参数签名中每个参数之间使用“#”进行分隔，取完一个方法所有的信息后，与类中的方法进行比较，如果完全匹配说明找到了需要Hook的方法，这个时候，调用`hookMethod()`方法进行Hook操作，注意这里的`hookMethod()`方法，即`Legend`框架支持的第二种Hook方式。

`hookMethod()`调用`Runtime.isArt()`判断当前代码执行在`Art`还是`Dalvik`模式，如果是`Art`模式，执行`hookMethodArt()`来完成Hook操作，如果是`Dalvik`模式，执行`hookMethodDalvik()`完成Hook。

`Runtime.isArt()`的代码只有一行，即判断虚拟机版本字符串是否以字符2开头，如下：

```
public static boolean isArt() {
    return getVmVersion().startsWith("2");
}

public static String getVmVersion() {
    return System.getProperty("java.vm.version");
}
```

执行完Hook后会返回一个`backupMethod`，这是一个原始方法的备份，最后将`backupMethod`放入以`methodName`命令的`backupList`，在`methodNameToBackupMethodsMap`备份就完事了。

接下来看看`hookMethodArt()`都干了啥：

```
private static Method hookMethodArt(Method origin, Method hook) {
        ArtMethod artOrigin = ArtMethod.of(origin);
        ArtMethod artHook = ArtMethod.of(hook);
        Method backup = artOrigin.backup().getMethod();
        backup.setAccessible(true);
```

```
        long originPointFromQuickCompiledCode =
artOrigin.getEntryPointFromQuickCompiledCode();
        long originEntryPointFromJni = artOrigin.getEntryPointFromJni();
        long originEntryPointFromInterpreter =
artOrigin.getEntryPointFromInterpreter();
        long originDeclaringClass = artOrigin.getDeclaringClass();
        long originAccessFlags = artOrigin.getAccessFlags();
        long originDexCacheResolvedMethods =
artOrigin.getDexCacheResolvedMethods();
        long originDexCacheResolvedTypes =
artOrigin.getDexCacheResolvedTypes();
        long originDexCodeItemOffset = artOrigin.getDexCodeItemOffset();
        long originDexMethodIndex = artOrigin.getDexMethodIndex();

        long hookPointFromQuickCompiledCode =
artHook.getEntryPointFromQuickCompiledCode();
        long hookEntryPointFromJni = artHook.getEntryPointFromJni();
        long hookEntryPointFromInterpreter =
artHook.getEntryPointFromInterpreter();
        long hookDeclaringClass = artHook.getDeclaringClass();
        long hookAccessFlags = artHook.getAccessFlags();
        long hookDexCacheResolvedMethods =
artHook.getDexCacheResolvedMethods();
        long hookDexCacheResolvedTypes = artHook.getDexCacheResolvedTypes();
        long hookDexCodeItemOffset = artHook.getDexCodeItemOffset();
        long hookDexMethodIndex = artHook.getDexMethodIndex();

        ByteBuffer hookInfo = ByteBuffer.allocate(ART_HOOK_INFO_SIZE);
        hookInfo.putLong(originPointFromQuickCompiledCode);
        hookInfo.putLong(originEntryPointFromJni);
        hookInfo.putLong(originEntryPointFromInterpreter);
        hookInfo.putLong(originDeclaringClass);
        hookInfo.putLong(originAccessFlags);
        hookInfo.putLong(originDexCacheResolvedMethods);
        hookInfo.putLong(originDexCacheResolvedTypes);
        hookInfo.putLong(originDexCodeItemOffset);
        hookInfo.putLong(originDexMethodIndex);

        hookInfo.putLong(hookPointFromQuickCompiledCode);
        hookInfo.putLong(hookEntryPointFromJni);
        hookInfo.putLong(hookEntryPointFromInterpreter);
        hookInfo.putLong(hookDeclaringClass);
        hookInfo.putLong(hookAccessFlags);
        hookInfo.putLong(hookDexCacheResolvedMethods);
        hookInfo.putLong(hookDexCacheResolvedTypes);
        hookInfo.putLong(hookDexCodeItemOffset);
        hookInfo.putLong(hookDexMethodIndex);
```

```
artOrigin.setEntryPointFromQuickCompiledCode(hookPointFromQuickCompiledCode);

artOrigin.setEntryPointFromInterpreter(hookEntryPointFromInterpreter);
        artOrigin.setDeclaringClass(hookDeclaringClass);
        artOrigin.setDexCacheResolvedMethods(hookDexCacheResolvedMethods);
        artOrigin.setDexCacheResolvedTypes(hookDexCacheResolvedTypes);
        artOrigin.setDexCodeItemOffset((int) hookDexCodeItemOffset);
        artOrigin.setDexMethodIndex((int) hookDexMethodIndex);

        int accessFlags = origin.getModifiers();
        if (Modifier.isNative(accessFlags)) {
            accessFlags &= ~ Modifier.NATIVE;
            artOrigin.setAccessFlags(accessFlags);
        }
        long memoryAddress = Memory.alloc(ART_HOOK_INFO_SIZE);
        Memory.write(memoryAddress,hookInfo.array());
        artOrigin.setEntryPointFromJni(memoryAddress);

        return backup;
    }
```

原方法与替换的方法分别为`artOrigin`与`artHook`，执行`artOrigin`的`backup()`完成方法的备份操作，`backup()`内部通过反射获取`AbstractMethod`类的`artMethod`字段，然后使用当前类的`method`进行填充，实际的操作就是复制一份当前类的`method`，此处不展开它的代码。

接下来的代码是获取`artOrigin`与`artHook`的重要字段，然后构造`ByteBuffer`类型的`hookInfo`，最后调用以下三行代码来完成Hook：

```
long memoryAddress = Memory.alloc(ART_HOOK_INFO_SIZE);
Memory.write(memoryAddress,hookInfo.array());
artOrigin.setEntryPointFromJni(memoryAddress);
```

`ArtMethod`在底层的内存结构定义仅次于Android源码的“art/runtime/art_method.h”文件，不同系统版本的Android这个结构体都可能会发现变化，为了保持兼容性，`Legend`在Java层手动定义保存了它们的字段偏移信息，
与“Legend/legendCore/src/main/java/com/lody/legend/art/ArtMethod.java”文件保存在同一目录，在调用`ArtMethod::of()`方法构造`ArtMethod`时，会根据不同的系统版本来构造不同的对象。

`Memory.write()`方法底层调用的`LegendNative.memput()`，它是一个native方法，对应的实现是`android_memput()`，代码如下：

```
// Legend/Native/jni/legend_native.cpp
void android_memput(JNIEnv * env, jclass clazz, jlong dest, jbyteArray src) {
    jbyte *srcPnt = env->GetByteArrayElements(src, 0);
    jsize length = env->GetArrayLength(src);
    unsigned char * destPnt = (unsigned char *)dest;
    for(int i = 0; i < length; ++i) {
        destPnt[i] = srcPnt[i];
    }
    env->ReleaseByteArrayElements(src, srcPnt, 0);
}
```

可以看出，馓的内存写操作是直接使用底层指定长度的字节流覆盖的，简单与暴力，而能够这样操作的原因，是当前操作的内存是自己的内存，想怎么干就怎么干！

`setEntryPointFromJni()`直接将原方法起始地址的指针内容，通过构造的`memoryAddress`覆盖写入！如此这般，`Art`模式下的Hook就完成了，当然，这其中很多小细节没有讲到，读者可以看行阅读它的代码。

接下来看看`Dalvik`下的Hook方法`hookMethodDalvik()`都干了啥：

```
// Legend/legendCore/src/main/java/com/lody/legend/HookManager.java
private static Method hookMethodDalvik(Method origin, Method hook) {
    DalvikMethodStruct dvmOriginMethod = DalvikMethodStruct.of(origin);
    DalvikMethodStruct dvmHookMethod = DalvikMethodStruct.of(hook);

    byte[] originClassData = dvmOriginMethod.clazz.read();
    byte[] originInsnsData = dvmOriginMethod.insns.read();
    byte[] originInsSizeData = dvmOriginMethod.insSize.read();
    byte[] originRegisterSizeData = dvmOriginMethod.registersSize.read();
    byte[] originAccessFlags = dvmOriginMethod.accessFlags.read();
    byte[] originNativeFunc = dvmOriginMethod.nativeFunc.read();

    byte[] hookClassData = dvmHookMethod.clazz.read();
    byte[] hookInsnsData = dvmHookMethod.insns.read();
    byte[] hookInsSizeData = dvmHookMethod.insSize.read();
    byte[] hookRegisterSizeData = dvmHookMethod.registersSize.read();
    byte[] hookAccessFlags = dvmHookMethod.accessFlags.read();
    byte[] hookNativeFunc = dvmHookMethod.nativeFunc.read();

    dvmOriginMethod.clazz.write(hookClassData);
    dvmOriginMethod.insns.write(hookInsnsData);
    dvmOriginMethod.insSize.write(hookInsSizeData);
    dvmOriginMethod.registersSize.write(hookRegisterSizeData);
    dvmOriginMethod.accessFlags.write(hookAccessFlags);

    ByteBuffer byteBuffer = ByteBuffer.allocate(DVM_HOOK_INFO_SIZE);
    byteBuffer.put(originClassData);
    byteBuffer.put(originInsnsData);
    byteBuffer.put(originInsSizeData);
    byteBuffer.put(originRegisterSizeData);
    byteBuffer.put(originAccessFlags);
    byteBuffer.put(originNativeFunc);
    //May leak
    long memoryAddress = Memory.alloc(DVM_HOOK_INFO_SIZE);
    Memory.write(memoryAddress, byteBuffer.array());
    dvmOriginMethod.nativeFunc.write(memoryAddress);
    return origin;
}
```

分析完`Art`模式，`Dalvik`下的就不难看懂的，`DalvikMethodStruct.of()`会返回`DalvikMethodStruct`类型结构体，它是`Dalvik`虚拟机内部`DalvikMethod`结构体的内线性布局表示。

`dvmOriginMethod`与`dvmHookMethod`分别代表原方法与Hook替换的方法，同样的，使用底层内存的写操作，对所有需要替换的字段进行替换。

最后就是Hook后的方法调用原方法了，它的代码如下：

```
// Legend/legendCore/src/main/java/com/lody/legend/HookManager.java
public <T> T callSuper(Object who, Object... args) {
    StackTraceElement[] traceElements =
Thread.currentThread().getStackTrace();
    StackTraceElement currentInvoking = traceElements[3];
    String invokingClassName = currentInvoking.getClassName();
    String invokingMethodName = currentInvoking.getMethodName();
    Map<String,List<Method>> methodNameToBackupMethodsMap =
classToBackupMethodsMapping.get(invokingClassName);
    if (methodNameToBackupMethodsMap != null) {
        List<Method> methodList =
methodNameToBackupMethodsMap.get(invokingMethodName);
        if (methodList != null) {
            Method method = matchSimilarMethod(methodList, args);
            if (method != null) {
                try {
                   if (Runtime.isArt()) {
                       return callSuperArt(method, who, args);
                   }else {
                       return callSuperDalvik(method, who, args);
                   }
                } catch (Throwable e) {
                    Logger.e("[---] Call super method with error : %s, detail
message please see the [Logcat :system.err].", e.getMessage());
                    e.printStackTrace();
                }
            }else {
                Logger.e("[---] Super method cannot found in backup map.");
            }
        }
    }
    return null;
}
```

这段代码是在之前保存的`methodNameToBackupMethodsMap`中查找备份的方法，找到后对`Art`与`Dalvik`模式分别调用`callSuperArt()`与`callSuperDalvik()`，前者比较简单，只是调用方法的`invoke()`就完事，而`Dalvik`模式由于没有像`Art`那样做备份，所以多出了一个字段回替换的操作，完事也是调用的`invoke()`来执行原方法。

0x4 一些感想

分析完上面的代码，可以出来`Legend`尽管实现了`Art`与`Dalvik`双模式下的Hook，但在实际逆向Hook中，还是有一些不足：

1. 不能Hook字段。在很多应用场景中可能会用到，这里有一个迂回的替代的方案是：在字段较敏感的方法中对方法做Hook，然后在Hook代码中反射操作字段。
2. Hook自定义的类加载器加载的类方法。由于反射查找的类的方法列表依赖于类的查找，对于部分自定义`ClassLoader`的情况，获取`Class`本身就存在着难度，更别说Hook它的方法了。

3. 兼容性。只支持4.2到6.0，当然，根据技术原理，从2.3到7.1应该都是可以做到的。
4. 稳定性。与该框架技术原理类似的还有很多，比较alibaba的`AndFix`，在系统自定义修改较多的情况下，框加要的稳定性存疑，当然，逆向工程时使用的稳定性远没有做产品要求的高，一些全新思路的Hook修改方案如`Tinker`可能也是一个不错的选择，留待以后测试了！

最后，讲完了它的原理，并没有讲如何在逆向工程中使用，这个交给聪明的安全研究人员作为思维发散。